Técnicas de Inteligência Artificial para

Programação de Jogos

Eloi L. Favero
Departamento de Informática
CCEN - UFPA

favero@ufpa.br

1

Sumário

Capítulo 1

Introdução ao Prolog

Alan Robinson (1965)
Resolução e unificação
|
R. Kowalski (1972)
Interpretação procedural
|
Alain Colmerauer (1973)
1ro Prolog
|
David Warren (1977)
Prolog Edinburgh

Objetivos do capítulo: Inicialmente apresentamos um histórico do Prolog. Em seguida, apresentamos uma introdução a linguagem Prolog definindo a nomenclatura básica, que inclui os conceitos de fato, regra, pergunta, cláusula, predicado, procedimento e programa.

1.1 Histórico

Um programa Prolog é uma coleção de fatos e de regras que definem relações entre os objetos do discurso do problema. Uma computação em Prolog envolve a dedução de conseqüências a partir das regras e fatos. O significado do programa é o conjunto de todas as conseqüências deduzíeis pela iterativa aplicação das regras, sobre os fatos iniciais e os novos fatos gerados. Portanto, Prolog é uma linguagem com uma fundamentação lógica, em teoria da prova.

A linguagem de programação Prolog se originou a partir de trabalhos de pesquisa em procedimentos de resolução para lógica de primeira ordem. Mais precisamente, o nascimento do Prolog aconteceu por volta de 1965, quando Alan Robinson desenvolveu os dois componentes chaves de um provador de teoremas para lógica de cláusulas (eqüivalente à lógica de primeira ordem):

- o procedimento de resolução;
- o algoritmo de unificação.

A partir deste resultado teórico, de 1965, várias tentativas foram feitas para desenvolver uma máquina de computação baseada no princípio de resolução. Isto só aconteceu na década de 70.

Por um lado, R. Kowalski, 1972, formulou uma interpretação procedimental para as cláusulas de Horn (uma versão restrita da lógica de cláusulas, para qual Robinson desenvolveu a resolução). Ele mostrou que uma cláusula (que eqüivale a uma implicação lógica), tal como

```
A if B1 and B2 and ... Bn
```

pode ser lida, e executada, como um procedimento em uma linguagem de programação recursiva, onde `A` é a cabeça do procedimento e os `Bi` são o seu corpo. Assim, o programa

```
prog if ler(Dado) and calcular(Dado, Result) and impr(Result)
```

é lido como: para executar `prog` executa-se `ler(Dado)` e então executa-se `calcular(Dado, Result)` e então executa-se `impr(Result)`. Esta leitura eqüivale a um procedimento imperativo:

```
procedure prog;
  begin
     ler(Dado);
     calcular(Dado, Result);
     impr(Result);
  end;
```

Esta interpretação de cláusulas é consistente com processo de resolução, onde a unificação é quem faz a manipulação dos dados: atribuição de valores, construção de estruturas de dados, seleção de dados e passagem de parâmetros. Aqui, em 1972, nasce a Programação em Lógica (PL).

Por outro lado, em 1973, Alain Colmerauer na Universidade de Aix - Marseille desenvolveu um provador teoremas para implementar sistemas de Processamento de Linguagem Natural, chamado **Prolog** (Programation et Logique), que empregava a interpretação de Kowalski.

Prolog vem de PROgramming in LOGic

Até a metade da década de 70, existia um consenso de que sistemas provadores de teoremas baseados em lógica eram irremediavelmente ineficientes. Em 1977, na Universidade de Edinburgh, David Warren conseguiu desenvolver um compilador eficiente para processar cláusulas de Horn, chamado **Prolog-10**, também conhecido como Prolog Edinburgh. O compilador de Warren é baseado em uma máquina abstrata hoje conhecida como WAM (Warren abstract machine) [1][1] que é a base de quase todas as implementações comerciais e acadêmicas hoje disponíveis.

Programação em Lógica vs. Prolog

Programação em Lógica (PL) não é sinônimo de Prolog. Inicialmente, em PL temos duas grandes classes de programas: cláusulas definidas (ou Horn) e não definidas. Prolog é um

[1] Também disponível para ser baixado da Web.

```
1  pai(tare, abraao).  %1
2  pai(tare, nacor).   %2
3  pai(tare, aran).    %3
4  pai(aran, lot).     %4   /* 7 fatos */
5  pai(aran, melca).   %5
6  pai(aran, jesca).   %6
7  mae(sara, isaac).   %7
8  %%
9      fem(X):-mae(X,Y).                  %1
10 irmao(X,Y):-pai(P,X),pai(P,Y), X=/=Y. %2   /* 3 regras */
11   tio(T,X):-pai(P,X),irmao(P,T).      %3
```

Figura 1.1: Programa da família bíblica

sistema que executa programas para cláusulas definidas. Por outro lado, muitos programas em Prolog não tem uma representação em PL pura, como veremos no próximo capítulo. Na prática, a Programação em Lógica como disciplina de fundamentos teóricos da computação deve seu sucesso ao Prolog.

Na seqüência aprestamos uma introdução ao Prolog. PL é retomada no próximo capítulo.

1.2 Programa = regras + fatos

Um programa Prolog é uma coleção de fatos e de regras que definem relações entre os objetos do discurso do problema. Uma computação em Prolog envolve a dedução de conseqüências a partir das regras e fatos. O **significado de um programa** é o conjunto de todos as conseqüências deduzíveis pela iterativa aplicação das regras, sobre os fatos iniciais e os novos fatos gerados.

1.2.1 Regras e fatos em Prolog

Um **programa** em Prolog é uma coleção de unidades lógicas chamadas de predicados. Cada **predicado** é uma coleção de cláusulas. Uma **cláusula** é uma regra ou um fato.

A figura 1.1 apresenta um programa em Prolog descrevendo a árvore genealógica de uma família bíblica, a família de Abraão. O programa descreve um conjunto de relações lógicas envolvendo os objetos do domínio do problema, que são as pessoas da família, referidas pelos seus nomes.

Segue uma descrição sintática dos elementos presentes no programa da Figura 1.1.

1.2.2 Termos e predicados

Em um programa em Prolog:

- Uma **variável** representa um elemento não especificado do domínio. Sintaticamente, uma variável sempre inicia com letra maiúscula;

- Uma **constante** representa um elemento específico do domínio. Pode ser númerica ou uma cadeia de caracteres (tipicamente, iniciando com letra minúscula).

No programa da família bíblica, são constantes todos os nomes das pessoas: `tare, abraao, nacor, sara, ...`; e, são variáveis: `X, Y, T, P.`

Regras e fatos, em Prolog, são representados por cláusulas definidas. Assim em Prolog, um programa é um conjunto de cláusulas. **Cláusulas** são agrupadas em predicados. Um **predicado** é definido por um conjunto de regras e fatos com o mesmo nome. Cláusulas são fórmulas lógicas construídas a partir de fórmulas atômicas:

- Se `P` é uma fórmula atômica então a sua negação também é uma fórmula, representada em Prolog por `\+ P`.

- Se `P` e `Q` são fórmulas então também a conjunção `P,Q`, disjunção `P;Q` e a condicional `P:-Q` são fórmulas.

Fórmulas formadas por operadores (`, ; :- \+`) são não atômicas.

Na verdade, `:-` representa uma implicação lógica escrita de forma invertida. Assim, `P:-Q` é equivalente a $Q \rightarrow P$, como veremos no próximo capítulo.

Fórmulas atômicas são construídas a partir de **termos**; Termos são construídos a partir de variáveis e constantes. Como segue:

- Toda constante é um **termo**; e, toda variável e um **termo**;

- Se $t_1, .., t_n$ são termos e f é um símbolo funcional então $f(t_1, .., t_n)$ também é um termo;

- Se $t_1, .., t_n$ são termos e p é um símbolo predicativo então $p(t_1, .., t_n)$ também é uma fórmula atômica ou predicado;

No programa da família bíblica, são exemplos de fórmulas: `pai(tare, nacor), fem(X), pai(P,X), ...`

Revisando o programa da família bíblica, podemos identificar duas partes, a primeira descrita por **fatos**, definidos pelas relações `pai/2` e `mae/2`; e, a segunda pelas **regras** `irmao/2`, `fem(inina)/1` e `tio/2`. `pai/2` denota o predicado `pai` com dois argumentos, enquanto que `fem/1` denota o predicado `fem` com um argumento. O predicado `pai/2` é definido por seis fatos. Em resumo, o programa da figura 1.1 é descrito por sete fatos e três regras que definem cinco predicados.

Termos definidos por símbolos funcionais, por exemplo, `idade(X)`, representam funções que retornam valores que são constantes no domínio do problema. Por exemplo, `idade(joao)` retornando 25, `sexo(joao)` retornando `masculino`. Idade e sexo são atributos ou qualidades de um indivíduo.

Termos e fórmulas atômicas possuem a mesma estrutura, mas uma fórmula é sempre um objeto lógico, isto é, retorna sempre um valor lógico (verdadeiro ou falso). Existe também uma diferença semântica entre um termo e uma fórmula. Um termo representa uma oração incompleta, ou um nominal, por exemplo, `idade(X), idade(joao)` pode ser lido como *A idade de X ..., A idade de Joao...".* Por outro lado, uma fórmula atômica representa sempre uma oração completa, por exemplo, a fórmula `idade(joao,23)` é lida como *A idade de João é 23;* e a fórmula `forte(joao)` é lida como *João é forte.*

Termos, Functores e Operadores

Um termo é um símbolo seguidos de n argumentos: $t(t_1, .., t_n)$. Nesta representação t é chamado de **functor**. Um functor pode ser um símbolo funcional ou um símbolo predicativo.

Quando $n = 0$ temos dois casos, se o símbolo é funcional temos uma constante que é um termo; e, se o símbolo for predicativo temos uma constante lógica que é uma fórmula atômica, por exemplo, `true` ou `false`.

Functores com um ou dois argumentos podem também ser escritos na forma de operadores. Por exemplo, `1+2` corresponde a `+(1,2)`, `-1` corresponde `-(1)`; e, `1=2` corresponde a `=(1,2)`. O uso de operadores em Prolog tem como objetivo tornar a escrita de termos e predicados mais econômica e fácil, utilizando menos parênteses e virgulas. Compare: `+(1,2)` com `1+2`, o primeiro utiliza três caracteres a mais, dois parênteses e uma vírgula.

1.2.3 Convenções para leitura de cláusulas

Para um fato com 2 parâmetros podemos ter duas leituras trocando-se de ordem os parâmetros. Seja o fato `pai(tare,abraão)` temos as leituras: *Taré é pai de Abraão* ou *Abraão é pai de Taré.* Preferimos a primeira forma, porém em Prolog não existe uma convenção padrão, fica a critério do programador.

A regra `fem(X):-mae(X,Y)` é lida como uma fórmula condicional: (*X é feminino*) **se** (*X é mãe de Y*). E, a regra `tio(T,X):-pai(P,X),irmao(P,T)` corresponde a (*P é tio de X*) **se** (*P é pai de X)e(P é irmão de T).*

Numa regra `cabeca:-corpo`, o corpo pode ser constituído de uma lista de predicados, ligados por vírgula, denotando uma conjunção lógica.

Abaixo, segue um lista de exemplos de leitura de cláusulas:

`pai(tare, abraao).`
leia-se *Tare é pai de Abraão*

`mae(sara, isaac).`
leia-se *Sara é mae de Isaac*

`fem(X):-mae(X,Y).`
leia-se (*X é feminina*) **se** (*X é mãe de Y*)

`irmao(X,Y):-pai(P,X),pai(P,Y), X=/=Y`
leia-se (*X é irmão de Y*) **se** *(P é pai de X) e (P é pai de Y) e (X≠Y)*

`tio(T,X):-pai(P,X),irmao(P,T).`
leia-se (*T é tio de X*) **se** *(P é pai de X) e (P é irmão de T)*

1.2.4 Perguntas

O significado de um programa é o conjunto de conseqüências que são deduzidas a partir dos fatos e regras. Assim sendo, um sistema Prolog (isto é, um sistema que executa um programa Prolog) pode ser visto como um interpretador de perguntas. Para sabermos se um fato arbitrário é conseqüência do programa perguntamos ao sistema, que responde Sim (Yes) se for verdade e Não (No) caso contrário. A figura 1.2 apresenta uma lista de perguntas que podemos fazer ao interpretador que executa o programa da figura 1.1.

Na primeira pergunta, `?-fem(sara)` *(Sara é feminina?)*, o sistema respondeu Sim. Para deduzir a resposta o sistema usou a regra 1 e o fato 7, como segue:

```
?- fem(sara).
   Yes.
?- fem(aran).
   No
?- fem(X).
   X=sara. Yes
?- irmao(melca, lot).
   Yes
?- irmao(lot,X).
   X=melca; X=jesca. Yes
?- tio(nacor,jesca)
   Yes
```

Figura 1.2: Perguntas sobre a família bíblica

`mae(sara, isaac)`	*Sara é mãe de Isaac*
`fem(X):-mae(X,Y)`	*(X é feminina)* **se** *(X é mãe de Y)*
`fem(sara)`	*Sara é feminina*

Note que este mesmo esquema responde a pergunta `?-fem(X)` (*Quem é feminina?*). O fato deduzido `fem(sara)` é comparado com o predicado da "pergunta" `fem(X)` onde se deduz que `X=sara`.

Do mesmo modo, o interpretador buscando conseqüências lógicas responde todas as perguntas da figura 1.2. A segunda pergunta é lida *Aran é feminino?*. A resposta é Não, pois não existem fatos do tipo `mae(Aron,X)`. Para esta dedução o Prolog assume a **hipótese do mundo fechado**: tudo o que não está descrito nos fatos ou não é deduzível dos fatos é falso. A terceira pergunta, *Quem é feminina?* o sistema retorna `X=sara`, dizendo que Sara é quem é feminina. Mais adiante, perguntamos *Quem é irmão de Lot?* `irmao(lot,X)` — o sistema dá duas respostas: Melca e Jesca. Ambos valores podem ser deduzidos para a variável `X`. De forma similar, o sistema responde logicamente para qualquer pergunta fórmulada sobre este programa.

Capítulo 2

Animação de programas

Por animação de programas, entendemos o estudo de técnicas que implementam uma interface amigável e intuitiva com o usuário. Estas técnicas permitem representar na tela do computador os modelos abstratos embutidos nos algoritmos. Por exemplo, a solução para o problema das torres de Hanoi tipicamente é uma lista de movimentos. Com animação busca-se simular os movimentos na tela do computador.

Neste capítulo trabalhamos com dois problemas: Torres de Hanoi e Jogo da Velha. No final do capítulo apresentamos alguns exercícios significativos, que servem para avaliar o aprendizado do aluno na programação em Prolog. A solução para o jogo da velha resulta em programas de aproximadamente 100 linhas de código. Estender as versões do jogo da velha, incluindo regras que definem novas estratégias de jogo é um trabalho significativo de programação.

2.1 Torres de Hanoi

Torres de Hanoi é um problema clássico usado para se apresentar programas recursivos. Este problema é atribuído a uma lenda que aconteceu em um mosteiro aos arredores de Hanoi. Deus, logo após ter criado o mundo, deu uma missão aos monges do mosteiro. E, disse que quando a missão estivesse concluída o mundo iria acabar.

A missão foi enunciada assim. Existem três hastes e na primeira delas existe uma torre com 64 discos de ouro. O objetivo é mover os 64 discos para a haste do centro usando a terceira haste como haste auxiliar. Na movimentação existem apenas duas regras:

- os discos são movidos de uma haste para outra, um a um; e,
- nunca um disco maior pode estar sobre um disco menor.

A melhor solução para um problema de N discos leva $2^N - 1$ movimentos. Portanto, se todos os movimentos forem corretos são necessários $2^{64} - 1 = 1.84467e + 19$ movimentos. Este número corresponde a mais de 18 quintiliões de movimentos. Isto é uma eternidade toda, não precisamos perder o sono preocupados com o fim do mundo. Os monges devem ainda estar trabalhando para cumprir a missão.

Para efeito de programação, o problema é enunciado para 2, 3 ou 5 discos. Abaixo, temos a solução para o problema com 2 discos, onde são usados $2^2 - 1 = 3$ movimentos.

```
?-hanoi(2).
  esq    centro   dir
```

```
   |       |       |        (Estado Inicial)
   =       |       |
  ===      |       |
"""""""" """""""" """""""""


   |       |       |
   |       |       |        (Move um disco da esq para a dir)
  ===      |       =
"""""""" """""""" """""""""


   |       |       |
   |       |       |        (Move um disco da esq para o centro)
   |      ===      =
"""""""" """""""" """""""""


   |       |       |
   |       =       |         (Move um disco da dir para o centro)
   |      ===      |
"""""""" """""""" """""""""
```

A estratégia para resolver o jogo consiste num procedimento `hanoi(N,A,B,C)` que move N discos de A(esq) para B(centro) usando C(dir) como haste auxiliar:

- se `N=0` então o problema está resolvido;
- senão mover N discos da haste A para a haste B usando a haste C como auxiliar é decomposto em:
  - mover M=N-1 discos da haste A para a haste C usando a haste B como auxiliar;
  - depois mover um disco (o do fundo da pilha) da haste A para B; e,
  - depois mover os M discos da haste C para a haste B usando a haste A como auxiliar.

Por ser um procedimento recursivo, ele só realmente executa o movimento base a cada vez que é chamado. O movimento base move um disco de uma pilha para outra. O procedimento inicial codificando esta estratégia da solução é apresentado abaixo.

```
1  hanoi0(N) :- write('Mover '), write(N),
2               write(' discos da Esq para o Centro'),nl,
3               write(' usando Dir como pilha auxiliar:'),nl,
4               moveXY0(N,esq,centro,dir).
5  moveXY0(0,_,_,_) :- !.
6  moveXY0(N,A,B,C) :- !, M is N-1,
7         moveXY0(M,A,C,B),
8         exibeMovXY0(A,B),get0(_),
9         moveXY0(M,C,B,A).
10 exibeMovXY0(X,Y) :- !,nl, write(' Move um disco de '), write(X),
11                           write(' para '), write(Y), nl.
```

Segue a execução deste procedimento para uma torre de três discos, onde são executados sete movimentos.

```
?- hanoi0(3).
 Mover 3 discos da Esq para o Centro
    usando Dir como pilha auxiliar:
 Move um disco de esq para centro
 Move um disco de esq para dir
 Move um disco de centro para dir
 Move um disco de esq para centro
 Move um disco de dir para esq
 Move um disco de dir para centro
 Move um disco de esq para centro
```

Nesta solução incluímos um comando `get0(_)` para rodar o programa passo a passo; após exibir o movimento, digitamos um `<enter>` para executar o próximo movimento. Nosso objetivo, agora, é estender esta solução para exibir passo a passo o movimento das torres como apresentado no início da seção.

2.1.1 Salvando e atualizando o estado das hastes

Para animar o procedimento temos que definir uma estrutura de dados para representar as três hastes. E, a cada movimento, por exemplo, *"Move um disco da esquerda para o centro"*, devemos atualizar a estrutura de dados. Além disso, devemos ter um predicado que exibe a estrutura de dados logo após ser atualizada.

Para salvar o estado do jogo, usamos dois fatos:

- `nElem(N)` salva o número de discos passados como parâmetro no jogo; e
- `estado(E,C,D)` salva o valor das três hastes.

Por exemplo, se chamamos `hanoi(3)` são criados os fatos: `nElem(3)` e `estado([1,2,3],[],[])`. A lista `[1,2,3]` representa um torre (pilha) de três discos sendo que o menor, o 1, está no topo e o maior, o 3, está no fundo da pilha. O primeiro movimento da solução *"Move um disco da esquerda para o centro"* atualiza o estado para `estado([2,3],[1],[])`.

Os fatos `nElem/1` e `estado/3` são declarados como dinâmicos, pois são atualizados dinamicamente durante a execução do programa, pelos predicados `mknElem/1` e `mkEstados/3`. Estes predicados criam um novo objeto com o parâmetro passado. Neles, o `retract` remove a versão velha e o `assert` cria a versão nova. Segue o código destes predicados.

```
1 :-dynamic(nElem/1).
2 :-dynamic(estado/3).
3 nElem(0).
4 estado([],[],[]).
5 mkEstado(E,C,D):- estado(X,Y,Z), retract(estado(X,Y,Z)), assert(estado(E,C,D)).
6 mknElem(N)     :- nElem(E),retract(nElem(E)), assert(nElem(N)).
```

É relativamente simples desenhar torres e hastes com um tamanho fixo, por exemplo, três. Porém, queremos desenhar torres de qualquer tamanho acima de um (dependendo dos limites da tela, até um máximo de 10 ou 15 discos). Para isso definimos alguns predicados de entrada e saída. O `writeN/2` escreve uma repetição do caracter passado como parâmetro — serve para escrever um disco de tamanho N. O predicado `wrtDisco` escreve um disco de tamanho D para uma haste de tamanho N ou apenas um elemento da haste. O predicado `wrtECD` escreve as três hastes com discos, linha por linha.

Para facilitar a escrita das hastes usamos um predicado `ajustaN`, que permite transformar um estado como `estado([1,2,3],[],[])` em um novo estado `estado([0,1,2,3], [0,0,0,0], [0,0,0,0])`. Esta representação do estado é usada só para o desenho: cada zero corresponde a um elemento de uma haste. Mesmo quando a pilha está cheia, desenhamos um elemento de haste sobre o topo, para parecer mais realista. O predicado `fazLista` é usado para inicializar a torre do jogo; por exemplo, para um jogo `hanoi(5)` é criada uma lista que representa uma torre com 5 discos `[1,2,3,4,5]`. Segue abaixo o código dos predicados que são usados no desenho do estado.

```
1  writeN(N,_) :- N<1,!.
2  writeN(N,C) :- write(C), N1 is N-1, writeN(N1,C).
3  %%
4  wrtDisco([0|Ds],N,Ds) :- T is (N*2+1) // 2 ,tab(T), write('|'), tab(T).
5  wrtDisco([D|Ds],N,Ds) :- T is (((N*2+1) - (D*2-1)) // 2), tab(T),
6                           writeN(D*2-1,'='), tab(T).
7  wrtDiscoECD(_,_,_,0,_) :- !.
8  wrtDiscoECD(E,C,D,N,M):- wrtDisco(E,M,E2), tab(1),
9                           wrtDisco(C,M,C2), tab(1),
10                          wrtDisco(D,M,D2), nl,
11                          N1 is N-1, wrtDiscoECD(E2,C2,D2,N1,M).
12 wrtHastes(E,C,D,T) :-
13     nl, ajustaN(E,T,E2), ajustaN(C,T,C2), ajustaN(D,T,D2),
14     wrtDiscoECD(E2,C2,D2,T,T), Z is T*2+1,
15     writeN(Z,'"'), tab(1), writeN(Z,'"'), tab(1),writeN(Z,'"'),nl.
16 %%
17 ajustaN(L,N,R) :- length(L,Tam), fazLista2(N-Tam,0,L/R).
18 fazLista2(N,_,Li/Li) :- N=<0, !.
19 fazLista2(N,V,Li/Lo) :- N1 is N-1, fazLista2(N1,V,[V|Li]/Lo).
20 %%
21 fazLista(M,L):-!, fazLista(M,1,L).
22 fazLista(M,N,[]) :- N>M,!.
23 fazLista(M,N,[N|R]) :- N1 is N+1, fazLista(M,N1,R).
```

Segue a execução de alguns dos predicados codificados acima: `writeN/2`, `ajustaN/3` e `fazLista/2`. A execução dos outros pode ser vista na animação apresentada no início da seção.

```
?- writeN(10,'=').
==========
?- ajustaN([1,2],4,L).
```

```
   L = [0, 0, 1, 2]
?- fazLista(5,L).
   L = [1, 2, 3, 4, 5]
```

O predicado `hanoi/1` define a versão animada do programa. Aqui, é necessário inicializar o estado do jogo e desenha-lo na tela. Este predicado chama o predicado principal do programa `moveXY`, que é praticamente o mesmo da versão inicial `moveXY0`. O que muda é a presença de um predicado `novoXYEst/2` que atualiza o estado antes de exibir o movimento com `exibeMovXY/2`.

O predicado `novoXYEst/2`, abaixo, está definido para todas as combinações de hastes (esquerda, centro, direita), duas a duas. Mover uma haste da esquerda para o centro centro é mover o valor da cabeça da lista da esquerda para a lista do centro, por exemplo, o `estado([3,4], [5], [1,2])` resulta no novo estado `estado([4],[3,5],[1,2])`.

```
1  hanoi(N) :- fazLista(N,L), mkEstado(L,[],[]), mknElem(N),
2               estado(Xs,Ys,Zs), write(' Estado Inicial'), nl,
3               wrtHastes(Xs,Ys,Zs,N+1), moveXY(N,e,c,d).
4  moveXY(0,_,_,_) :- !.
5  moveXY(N,A,B,C) :- !, M is N-1,
6         moveXY(M,A,C,B),
7         novoXYEst(A,B), exibeMovXY(A,B),get0(_),
8         moveXY(M,C,B,A).
9  exibeMovXY(X,Y) :- !, nl, write(' Move um disco de '),
10                  write(X),  write(' para '), write(Y), nl,
11                      estado(Xs,Ys,Zs), nElem(T), wrtHastes(Xs,Ys,Zs,T+1).
12 novoXYEst(e,c) :- estado([X|Xs],Ys,Zs), mkEstado(Xs,[X|Ys],Zs).
13 novoXYEst(e,d) :- estado([X|Xs],Ys,Zs), mkEstado(Xs,Ys,[X|Zs]).
14 novoXYEst(d,e) :- estado(Xs,Ys,[Z|Zs]), mkEstado([Z|Xs],Ys,Zs).
15 novoXYEst(d,c) :- estado(Xs,Ys,[Z|Zs]), mkEstado(Xs,[Z|Ys],Zs).
16 novoXYEst(c,d) :- estado(Xs,[Y|Ys],Zs), mkEstado(Xs,Ys,[Y|Zs]).
17 novoXYEst(c,e) :- estado(Xs,[Y|Ys],Zs), mkEstado([Y|Xs],Ys,Zs).
```

Este exemplo ilustra uma solução representando cada estado de um problema do tipo transição de estados — um estado é mantido e atualizado a cada passo ou transição. Este exemplo ilustra, também, como controlar os movimentos, passo a passo, com o uso de um comando de leitura o `get0/1` (que lê qualquer caractere, mesmo os de controle que não são visíveis na tela).

Na versão inicial, a solução do problema é simples e compacta (são 11 linhas de código). Por outro lado, a versão com animação na tela o código cresceu para várias vezes o tamanho do código inicial:

- seis linhas no fragmento 1 – salva o estado;
- mais 23 linhas fragmento 2 – escreve os discos;
- mais 17 linhas do fragmento principal;
- juntando os três fragmentos dá um total 46 linhas.

Agora podemos falar sobre animação de programas, que é o título deste capítulo, como uma técnica que permite visualizar na tela do computador, de uma forma amigável, o comportamento de um programa. Esta técnica tem várias aplicações uma delas pode ser para ensino de algoritmos, outra pode ser para programas como jogos interativos onde o computador faz o papel de um dos jogadores; por exemplo, num jogo de xadrez entre o usuário do computador.

Na próxima subseção, apresentamos a animação do programa para o jogo da velha.

2.2 Jogo da Velha: Usando um termo (JV-termo)

O jogo da velha consiste em um tabuleiro de 3x3, como mostrado abaixo. É um jogo entre dois adversários: normalmente um deles marca uma "cruz" e o outro marca uma "bola" em uma das casas do tabuleiro. Um dos jogadores começa o jogo marcando o tabuleiro e em seguida passa a vez ao adversário. O jogo termina com um vencedor ou com empate. Um jogador vence quando faz três "cruzes" (ou bolas) alinhadas, no sentido horizontal, vertical ou diagonal. Quando já não existe possibilidades de alguém fazer três marcações alinhadas o jogo empata. Seguem alguns exemplos de marcações de estados de jogos.

```
%            exemplo:   vence:   empates:
%  1|2|3      x| |      o|o|x    x|o|x   x|x|o
%  ------     -----     -----    -----   -----
%  4|5|6       |o|      x|o|     x|o|    o|o|x
%  ------     -----     -----    -----   -----
%  7|8|9       | |      x|x|o    o|x|o   x| |
```

Aqui, estudamos uma versão do jogo onde um dos jogadores é o computador. Vamos fazer duas versões: a primeira usando um termo como estrutura de dados para representar o tabuleiro e a segunda usando uma lista como estrutura de dados.

Na primeira versão, a representação do tabuleiro é um termo com nove argumentos onde cada argumento é uma casa do tabuleiro. Uma posição não marcada (vazia) é representada por uma variável livre. Quando uma posição é marcada: o valor cruz ("x") ou bola ("o") é unificado com a variável livre que está na posição correspondente.

Quatro predicados são usados para se testar a situação de uma posição no tabuleiro: `cruz(N,Tab)`, `bola(N,Tab)`, `vazio(N,Tab)` e `cheio(N,Tab)`. Estes predicados são definidos a partir da primitiva `arg(N,Termo,V)`, que retorna (ou unifica) o argumento número N de um Termo, como o valor V. Este valor pode ser qualquer objeto, inclusive uma variável. Este predicado é também usado para se realizar a marcação de uma posição no tabuleiro (foi renomeado como `moveC/3`).

Os predicados `var/1` e `nonvar/1` são usados, respectivamente, para se testar se um termo é ou não uma variável livre. Por exemplo, o predicado `cruz/2`, abaixo, é lido como: se o argumento N de Tab(uleiro) não é uma variável e é igual a "x" então temos uma cruz na posição N. O predicado `vazia/2` é verdadeiro se a posição N é uma variável livre.

```
1 %% tab(_,_,o, _,_,_, _,_,_)  % tabuleiro como um termo
2 %% tab(1 2 3  4 5 6  7 8 9)
3 %%
4 moveC(N,Tab,C):- arg(N,Tab,C).
5  cruz(N,Tab)  :- arg(N,Tab,V), nonvar(V), V=x.
6  bola(N,Tab)  :- arg(N,Tab,V), nonvar(V), V=o.
```

```
7 vazia(N,Tab)  :- arg(N,Tab,V), var(V).
8 cheia(N,Tab)  :- \+ vazia(N,Tab).
```

Abaixo exemplificamos o uso destes predicados.

```
?- T=tab(_,_,o, _,_,_, _,_,_), vaziaN(3,T).
   No
?- T=tab(_,_,o, _,_,_, _,_,_), cheia(3,T).
   Yes
?- T=tab(_,_,o, _,_,_, _,_,_), bola(3,T).
   Yes
?- T=tab(_,_,o, _,_,_, _,_,_), cruz(3,T).
   No
```

Um jogador vence quando, após um movimento, três marcações iguais (cruzes ou bolas) estão alinhadas. Para saber se três marcações estão alinhadas temos o predicado `emlinha3`, que é definido para as posições horizontais (3 possibilidades), verticais (3 possibilidades) e diagonais (2 possibilidades).

```
1  emlinha3([1,2,3]). %% horiz  %% posições em linha
2  emlinha3([4,5,6]).
3  emlinha3([7,8,9]).
4  emlinha3([1,4,7]). %% vert
5  emlinha3([2,5,8]).
6  emlinha3([3,6,9]).
7  emlinha3([1,5,9]). %% diag
8  emlinha3([3,5,7]).
9  %%
10 vence(T,venceu(cruz)):- emlinha3([A,B,C]), cruz(A,T),cruz(B,T),cruz(C,T),!.
11 vence(T,venceu(bola)):- emlinha3([A,B,C]), bola(A,T),bola(B,T),bola(C,T),!.
```

O predicado `vence/2` testa se três valores em linha são marcados com cruz ou bola. Ele devolve um termo indicando quem venceu: `venceu(cruz)` ou `venceu(bola)`. Poderíamos devolver só o valor `cruz` ou `bola`, mas o termo carrega uma informação mais completa: na pergunta "X vence?", ele responde "Cruz venceu".

É necessário um predicado para saber se o jogo terminou: "game over". O jogo termina se um dos jogadores vence ou numa situação de empate. É empate quando nenhum dos dois jogadores pode ganhar. Isto significa que, dada um estado do jogo, se preenchermos as posições livres com bolas (ou com cruzes) não teremos 3 bolas (ou cruzes) em linha.

Para definirmos o `empate` usamos o predicado `preenche` que sistematicamente seleciona uma posição vazia (1..9) e preenche a posição com bola (ou cruz). O predicado termina quando não tem mais posições vazias.

```
1 preenche(XO,T):- member(X,[1,2,3,4,5,6,7,8,9]),
2                  vazia(X,T),moveC(X,T,XO),!,preenche(XO,T).
3 preenche(XO,T).
4 empate(T):- preenche(o,T),\+ vence(T,_),!,preenche(x,T),\+ vence(T,_).
```

Abaixo chamamos o predicado `empate`. A primeira pergunta verifica que não é empate um jogo num tabuleiro com apenas duas posições preenchidas. A Segunda pergunta verifica o empate num tabuleiro com apenas uma posição livre. Note que após um teste de empate o tabuleiro fica totalmente preenchido. Quando o empate falha, o tabuleiro permanece como estava. Sempre que um predicado falha, tudo o que foi feito sobre os dados dos seus parâmetros é desfeito.

```
?- (T=tab(o,_,x,  _,_,_,  _,_,_), empate(T)).
   No
?- (T=tab(x,o,x,  x,o,_,  o,x,o), empate(T)).
   T = tab(x,o,x,x,o,o,o,x,o)
   Yes
```

2.2.1 Estratégia de jogo para vencer (ou não perder)

Mesmo num jogo simples como o jogo-da-velha é necessário usar uma estratégia de jogo para não se perder o jogo pelo adversário. Por exemplo, suponha que eu jogo com a cruz e é a minha vez de jogar:

- primeiro, se já tenho duas cruzes alinhadas e a terceira posição da linha está livre, então devo jogar nesta posição para ganhar (prioridade um é ganhar);
- segundo, se o adversário tem duas bolas alinhadas e a terceira está livre, tenho que me defender, jogando na posição livre para ele não ganhar (prioridade dois, não perder na próxima jogada);
- terceiro, se não posso ganhar e nem preciso me defender, então devo escolher a melhor posição para a jogada: por exemplo, primeiro vejo se a posição 5 (do centro) está livre; se sim, jogo nela; senão, vejo se um canto está livre e jogo nele; senão, jogo em qualquer posição livre.

O primeiro e segundo casos são similares. Eles podem ser programados com um predicado que verifica se existe uma ameaça (dois em linha, com a terceira posição vazia). Existem dois casos para o predicado `ameaca`: um para bola e outro para cruz.

```
1 ameaca(Tab,CB,W) :- emlinha3(Pos),ameaca(CB,Pos,Tab,W),!.
2 %%
3 ameaca(cruz,[A,B,C],T,A) :- vazio(A,T),cruz(B,T),cruz(C,T).
4 ameaca(cruz,[A,B,C],T,B) :- vazio(B,T),cruz(A,T),cruz(C,T).
5 ameaca(cruz,[A,B,C],T,C) :- vazio(C,T),cruz(A,T),cruz(B,T).
6 %%
7 ameaca(bola,[A,B,C],T,A) :- vazio(A,T),bola(B,T),bola(C,T).
8 ameaca(bola,[A,B,C],T,B) :- vazio(B,T),bola(A,T),bola(C,T).
9 ameaca(bola,[A,B,C],T,C) :- vazio(C,T),bola(A,T),bola(B,T).
```

Seguem dois testes para o predicado que verifica uma ameaça.

```
?- T=tab(_,o,o, _,x,_, _,_,_),ameaca(T,cruz,P).
   No
```

```
?- T=tab(_,o,o, _,x,_, _,_,_),ameaca(T,bola,P).
   T = tab(_G399, o, o, _G402, x, _G404, _G405, _G406, _G407)
   P = 1 ;
   No
```

No predicado `escolheMov/2` passamos o tabuleiro e o jogador; ele retorna um novo tabuleiro com a jogada feita pelo jogador (computador ou oponente). Não é necessário passar um tabuleiro de entrada e outro de saída, porque as posições livres do tabuleiro são variáveis. Assim, fazer uma jogada numa determinada posição consiste em unificar uma variável com um valor.

Agora, estamos em condições de codificar a estratégia de jogo exposta acima — no predicado `escolheMov`, segue abaixo. Assumimos que o computador joga com bola e que o adversário joga com cruz.

```
1 escolheMov(T, computador):-
2       ameaca(T,bola,W),!,moveC(W,T,o),! %% vence
3     ; ameaca(T,cruz,W),!,moveC(W,T,o),! %% defesa
4     ; vazia(5,T),moveC(5,T,'o'),!
5     ; chute(9,W),member(W,[1,3,7,9]),vazia(W,T),moveC(W,T,'o'),!
6     ; chute(9,W),member(W,[2,4,6,8]),vazia(W,T),moveC(W,T,'o'),!.
```

Para saber se o computador pode vencer na próxima jogada perguntamos `ameaca(T,bola,W)`, onde T é o tabuleiro e W é a posição em que acontece a ameça. Se o predicado `ameaca` for verdadeiro então usamos o predicado `moveC(W,T,o)` para jogar uma bola na posição W. Se não podemos vencer, temos que verificar se o adversário está nos ameaçando, com `ameaca(T,cruz,W)` Neste caso, também, temos que jogar na posição W.

Caso não podemos vencer e nem estamos ameaçados então tentamos jogar no centro; senão num dos cantos (posições 1,3,7,9); e, senão numa outra posição qualquer (2,4,6,8). Para jogar bola num canto, podemos fazer a pergunta abaixo.

```
?- member(W,[1,3,7,9]),vazio(W,T),moveC(W,T,'o').
```

Aqui o predicado member seleciona em W um dos valores dos cantos da lista [1,3,7,9]; se o canto W está vazio joga-se em W; caso contrario, por retrocesso, é escolhido o próximo valor e o processo se repete. Se nenhum das posições dos valores da lista está livre a pergunta falha. Este mesmo processo é válido para jogar nas posições pares [2,4,6,8]; o que muda é a lista com as posições.

Esta solução para jogar nos cantos funciona. Mas ela sempre escolhe o primeiro valor da lista dos cantos. Uma solução mais inteligente pode escolher um valor da lista aleatoriamente a cada jogada. Para isso devemos podemos usar um predicado `chute` que funciona como segue.

```
?- chute(5,X).
   X = 5;  X = 1;  X = 4 ;  X = 2 ; X = 2 ; X = 1 ; ...
```

Este predicado chute é implementado a partir de um predicado que gera números aleatórios. Aqui codificamos o predicado `randomico` (o SWI-Prolog possui a função predefinida `random/1`). Para fazer retrocesso no chute, gerando sucessivos chutes (por exemplo, até acertar um dos cantos), usa-se o predicado `repeat`. Sem o `repeat` só um valor de chute é gerado; e, se a posição deste valor não estiver vazia ou não estiver na lista dos cantos o corpo da cláusula que escolhe o movimento falha.

```
chute(N,S) :- repeat, randomico(N,S).
%chute(N,S):- repeat, S is random(N)+1. % SWI-prolog
```

O predicado `randomico`, que gera valores aleatórios, pode ser codificado com o auxilio de um termo que salva uma semente de geração `seed_` e uma função matemática para o cálculo do próximo valor aleatório a partir da derradeira[1] semente. O predicado abaixo é baseado na versão apresentada no livro de Clocksin e Mellish [4].

```
:-dynamic(seed_/1).  seed_(13).
randomico(N,R):- seed_(S), retract(seed_(S)),
                 R is ( S mod N ) + 1,
                 NS is (125 * S + 1) mod 4096,
                 assert(seed_(NS)),!.
```

2.2.2 O jogo do adversário do computador

Um jogo entre um computador e um adversário (ou oponente) consiste em um ciclo de rodadas, onde um jogador inicia o jogo e passa a vez ao seu adversário. Na tela do computador a cada jogada um novo tabuleiro deve ser exibido, considerando o derradeiro movimento.

Segue o predicado principal `jogo` que inicializa um tabuleiro vazio, exibe o tabuleiro e chama o oponente do computador para jogar, com o predicado `jogar/2`. Este predicado, recursivo na cauda, executa o ciclo de jogadas: a cada jogada o parâmetro jogador é trocado com o predicado `proximo`: o próximo do computador é o oponente e vice-versa. Em cada ciclo existe a chamada ao predicado `escolheMov` que é chamado para o jogador da vez. Após a escolha do movimento é exibido o tabuleiro com o estado atualizado. Ao mesmo tempo, após cada jogada é verificado se o jogo terminou com o predicado `gameOver`. O jogo termina com um vencedor ou com empate. Os predicados `vence` e `empate` já foram comentados. O pedicado `gameOver` também retorna um resultado que pode ser, `venceu(bola)`, `venceu(cruz)` ou `empate`.

```
jogo :- T = tab(A,B,C,  D,E,F,   G,H,I),
        exibeJogo(T, inicio),
        jogar(T, oponente).
jogar(T, Jogador):- gameOver(T,Result),!,msgFim(Result).
jogar(T, Jogador):- escolheMov(T, Jogador),!,
                    exibeJogo(T, Jogador),!,
                    proximo(Jogador, Oponente), !,
                    jogar(T, Oponente).
proximo(computador,oponente).
proximo(oponente,computador).
%%
exibeJogo(T,J):- write('jogou:'),write(J),desenha(T).
msgFim(X):-write('GAME OVER:'), write(X),nl,nl.
%%
```

[1] Palavra da língua Portuguesa pouco usada, mas necessária. Com significado similar a *última*; significando a "última por enquanto"; se fosse a última mesmo não poderia ter uma próxima.

```
15 gameOver(T,V) :- vence(T,V).
16 gameOver(T,empate) :- empate(T).
```

Par completar apresentação do predicado `escolheMov` é necessário descrever como é feita uma jogada para o adversário do computador. O predicado `escolheMov(T,oponente)` testa com o predicado `vaziaN` se a posição lida corresponde a uma posição vazia no tabuleiro; se este teste falha, uma mensagem informa que a jogada não é válida; e, é repetida a chamada ao predicado de escolha do movimento.

```
1 testaOk(P,Tab) :- vazia(P,Tab),arg(P,Tab,'x'),!;
2                   write('Jogada invalida,tente outra!'),nl,
3                   escolheMov(Tab,oponente).
4 escolheMov(T, oponente):- write('jogue (1..9):'),nl,
5                           read(P), testaOk(P,T).
```

2.2.3 Desenhar o tabuleiro

Finalmente, falta descrever o predicado que desenha o tabuleiro. Desenhar o tabuleiro a partir do termo `tab/9` não apresenta maiores dificuldades. Para escrever cada linha do tabuleiro definimos um predicado `wrtLinha` onde são passados como parâmetros os valores para as linhas horizontais (1,2,3), (4,5,6) e (7,8,9).

```
1 %% desenha o tabuleiro
2 wrtLinha(X,Y,Z,T):-arg(X,T,V1), wVal(V1),write('|'),
3                    arg(Y,T,V2), wVal(V2),write('|'),
4                    arg(Z,T,V3), wVal(V3),nl.
5 wVal(X):- var(X)->write(' ');write(X).
6 desenha(T) :- nl, tab(7),wrtLinha(1,2,3,T), tab(7),write('------'),nl,
7                   tab(7),wrtLinha(4,5,6,T), tab(7),write('------'),nl,
8                   tab(7),wrtLinha(7,8,9,T).
```

Segue a execução do predicado `desenha`.

```
?-T=tab(_,_,o, _,x,_, _,_,_), desenha(T).
      | |o
     ------
      |x|
     ------
      | |
```

Note que, diferente do problema das Torres de Hanoi apresentado na seção anterior, o jogo da velha não guarda o estado de cada jogada como um termo no banco de dados do Prolog. Sempre que trabalhamos com um ciclo de transições é possível fazer a escolha entre: (1) usar o estado como um parâmetro a ser passado entre os predicados do programa; ou (2) usar um termo global armazenado no banco de dados do Prolog.

Nesta versão do jogo usamos um parâmetro nos predicados para representar o estado do jogo. A cada jogada uma variável livre do termo é substituída por um valor. No final do jogo o

termo está quase cheio — quase porque o jogo normalmente termina antes de se preencher todas as casas. Por ser um termo com variáveis livres foi necessário apenas passar como parâmetro de entrada o tabuleiro.

Ao contrário, no problema das torres a cada movimento um novo estado é criado. Portanto, para fazer uma versão do problema das torres sem usar o banco de dados é necessário passar o estado como entrada e também como saída, nos predicados que executam os movimentos. Entra o estado atual e sai o novo estado: por exemplo, entra `estado([1,2],[],[])` e sai `estado([2],[1],[])`.

2.2.4 A listagem do programa todo

Vale a pena ver o código do programa como um todo. Juntando todos os fragmentos de código em um programa obtemos um texto com aproximadamente 90 linhas de código fonte. Destas mais que um quarto e menos que um terço são linhas de comentários.

Note também que no meio do programa temos várias perguntas que servem para testar os predicados codificados. Normalmente estas perguntas estão próximas do código do predicado; são exemplos de usos do predicado.

```
1  %% Jogo da Velha: (versão JV-termo)
2  %% ----------------------------------------------------------
3  %% tab(_,_,o, _,_,_, _,_,_)  % tabuleiro como um termo
4  %% tab(1 2 3  4 5 6  7 8 9)
5  %%
6  emlinha3([1,2,3]). %% horiz  %% posições em linha
7  emlinha3([4,5,6]).
8  emlinha3([7,8,9]).
9  emlinha3([1,4,7]). %% vert
10 emlinha3([2,5,8]).
11 emlinha3([3,6,9]).
12 emlinha3([1,5,9]). %% diag
13 emlinha3([3,5,7]).
14 %%
15 moveC(N,Tab,C):- arg(N,Tab,C).
16  cruz(N,Tab)  :- arg(N,Tab,V), nonvar(V), V=x.
17  bola(N,Tab)  :- arg(N,Tab,V), nonvar(V), V=o.
18 vazia(N,Tab)  :- arg(N,Tab,V), var(V).
19 cheia(N,Tab)  :- \+ vazia(N,Tab).
20 %%---------------------------------------------------------
21 gameOver(T,V) :- vence(T,V).
22 gameOver(T,empate) :- empate(T).
23 %%
24 vence(T,venceu(cruz)):- emlinha3([A,B,C]), cruz(A,T),cruz(B,T),cruz(C,T),!.
25 vence(T,venceu(bola)):- emlinha3([A,B,C]), bola(A,T),bola(B,T),bola(C,T),!.
26 %%
27 preenche(XO,T):- member(X,[1,2,3,4,5,6,7,8,9]),
28                  vazia(X,T),moveC(X,T,XO),!,preenche(XO,T).
29 preenche(XO,T).
```

```
30 empate(T):- preenche(o,T),\+ vence(T,_),!,preenche(x,T),\+ vence(T,_).
31 %% ?- (T=tab(o,_,x, _,_,_ ,_,_,_), empate(T)).
32 %% ?- (T=tab(o,o,x, x,x,o,o,o,x), empate(T)).
33 %%---------------------------------------------------------------------
34 testaOk(P,Tab) :- vazia(P,Tab),arg(P,Tab,'x'),!;
35                   write('Jogada invalida,tente outra!'),nl,
36                   escolheMov(Tab,oponente).
37 escolheMov(T, oponente):- write('jogue (1..9):'),nl,
38                           read(P), testaOk(P,T).
39 escolheMov(T, computador):-
40      ameaca(T,bola,W),!,moveC(W,T,o),! %% vence
41    ; ameaca(T,cruz,W),!,moveC(W,T,o),! %% defesa
42    ; vazia(5,T),moveC(5,T,'o'),!
43    ; chute(9,W),member(W,[1,3,7,9]),vazia(W,T),moveC(W,T,'o'),!
44    ; chute(9,W),member(W,[2,4,6,8]),vazia(W,T),moveC(W,T,'o'),!.
45 %%
46 ameaca(Tab,CB,W) :- emlinha3(Pos),ameaca(CB,Pos,Tab,W),!.
47 %%
48 ameaca(cruz,[A,B,C],T,A) :- vazia(A,T),cruz(B,T),cruz(C,T).
49 ameaca(cruz,[A,B,C],T,B) :- vazia(B,T),cruz(A,T),cruz(C,T).
50 ameaca(cruz,[A,B,C],T,C) :- vazia(C,T),cruz(A,T),cruz(B,T).
51 %%
52 ameaca(bola,[A,B,C],T,A) :- vazia(A,T),bola(B,T),bola(C,T).
53 ameaca(bola,[A,B,C],T,B) :- vazia(B,T),bola(A,T),bola(C,T).
54 ameaca(bola,[A,B,C],T,C) :- vazia(C,T),bola(A,T),bola(B,T).
55 %%-------------------------------------------------------------
56 %% desenha o tabuleiro
57 wrtLinha(X,Y,Z,T):-arg(X,T,V1), wVal(V1),write('|'),
58                     arg(Y,T,V2), wVal(V2),write('|'),
59                     arg(Z,T,V3), wVal(V3),nl.
60 wVal(X):- var(X)->write(' ');write(X).
61 desenha(T) :- nl, tab(7),wrtLinha(1,2,3,T), tab(7),write('------'),nl,
62                   tab(7),wrtLinha(4,5,6,T), tab(7),write('------'),nl,
63                   tab(7),wrtLinha(7,8,9,T).
64 %% ?- T=tab(_,_,o, _,x,_, _,_,_), desenha(T).
65 %%-------------------------------------------------------------
66 %% esquema principal do jogo
67 %%
68 jogo :- T = tab(A,B,C,  D,E,F,   G,H,I),
69         exibeJogo(T, inicio),
70         jogar(T, oponente).
71 jogar(T, Jogador):- gameOver(T,Result),!,msgFim(Result).
72 jogar(T, Jogador):- escolheMov(T, Jogador),!,
73                     exibeJogo(T, Jogador),!,
74                     proximo(Jogador, Oponente), !,
75                     jogar(T, Oponente).
76 proximo(computador,oponente).
```

```
77 proximo(oponente,computador).
78 %%
79 exibeJogo(T,J):- write('jogou:'),write(J),desenha(T).
80 msgFim(X):-write('GAME OVER:'), write(X),nl,nl.
81 %%
82 %%-------------------------------------------------------------
83 chute(N,S) :- repeat, randomico(N,S). %%% repeat = backtracking
84 chute2(N,S):-repeat, S is random(N). %% swi-prolog
85 :-dynamic(seed_/1).  seed_(13).
86 randomico(N,R):- seed_(S), retract(seed_(S)),
87                      R is ( S mod N ) + 1,
88                      NS is (125 * S + 1) mod 4096,
89                      assert(seed_(NS)),!.
90 %% ?- randomico(5,X).
```

2.3 Projetos: Utilizando estratégias de jogo

A estratégia de jogo codificada nestas versões do jogo da velha ainda é pobre, enriquece-la cabe ao leitor. Por exemplo, sabemos que se um jogador parte jogando na posicão central e o oponente não joga num canto, o oponente perde o jogo. A versão acima não codifica esta estratégia. Veja abaixo o que acontece:

```
computador bola
movimentos:     (1) (2) (3) (4) (5) (6) (7) (8)
                5o, 2x, 1o, 9x, 3o, 7x, 8o, 4x,

                +++ +x+ ox+ ox+ oxo oxo oxo oxo
                +o+ +o+ +o+ +o+ +o+ +o+ +o+ xo+  empate
                +++ +++ +++ ++x ++x x+x xox xox
```

Neste caso, pode-se usar uma "armadilha": o computador no quinto movimento jogando na posição sete cria duas linhas com ameaça da bola e a cruz perde o jogo. Veja abaixo.

```
computador bola
movimentos: (1) (2) (3) (4) (5) (6) (7)
            5o, 2x, 1o, 9x, 7o, 3x, 4o,

            +++ +x+ ox+ ox+ ox+ oxx oxo
            +o+ +o+ +o+ +o+ +o+ +o+ oo+  venceu bola
            +++ +++ +++ ++x o+x o+x oox
```

Note, esta estratégia pode ser generalizada: se no segundo movimento o oponente não jogar num canto ele perde, não importa em qual das quatro posições ele joga.

Outra estratégia está relacionada com os cantos. Como ilustrado abaixo, a idéia é encontrar uma posição comum livre em duas linhas, onde cada linha tem uma marcação bola e duas livres. Se jogamos a marcação comum criamos uma armadilha.

```
computador bola
movimentos:    (1) (2) (3) (4) (5) (6) (7)
               1o, 5x, 9o, 3x, 7o, 4x, 8o,

               o++ o++ o++ o+x o+x o+x o+x
               +++ +x+ +x+ +x+ +x+ xx+ xx+  venceu bola
               +++ +++ ++o ++o o+o o+o ooo
```

Exercício 2.1 Programe as armadilhas descritas acima para o jogo do computador ficar mais inteligente.

Exercício 2.2 Faça uma versão do jogo onde pode-se escolher quem começa. Quando o oponente começa certifique-se que o computador nunca perde.

Exercício 2.3 Faça um versão do jogo que o computador começa de uma posição aleatória.

Exercício 2.4 Faça uma versão do jogo, que simule a inteligência de uma criança: jogando em posições aleatórias; as vezes pensando e se defendendo; e, as vezes caindo numa armadilha. O oponente pode armar armadilhas e vencer o jogo.

Capítulo 3

Técnicas de Busca e Jogos

A área de Inteligência Artificial (IA) ou Computação Inteligente (CI) trata basicamente de problemas para os quais ainda não temos soluções que podem ser expressas em forma de um algoritmo já estabelecido (já conhecido e estudado). Muitos dos problemas para os quais não temos algoritmos já estabelecidos, podem ser resolvidos por uma modelagem que cria um espaço de busca que contém as soluções. Algoritmos especiais, guiados por regras heurísticas, procuram as soluções nestes espaços.

Existem dois métodos gerais de caminhamento em árvore: profundidade e largura. A busca *profundidade* executa um caminhamento sobre a árvore priorizando entrar numa árvore filha (de pai para filho) em relação ao processamento dos nós irmãos; a busca em *largura* executa um caminhamento priorizando buscas nos nós irmãos.

Noutra perspectiva, um caminhamento pode ser de dois tipos(ver capítulo sobre estruturas de dados):

- *exaustivo*: visita-se todos os nós da árvore; por exemplo, um caminhamento em pré-ordem.
- *não-exaustivo*: visita-se somente parte dos nós, preferencialmente uma pequena parte; por exemplo, a busca de um valor numa arvore binária de busca.

Tipicamente, um espaço de busca é muito grande par ser explicitamente construído e exaustivamente examinado na busca de soluções. Portanto, em problemas reais o processo de busca é guiado por um conjunto de regras de heurísticas que visam otimizar o tempo da busca.

Neste capítulo, começamos apresentando os métodos básicos de busca, em largura e profundidade. Em seguida, apresentamos métodos de busca heurística que estendem os métodos básicos de busca com o uso de regras heurísticas que visam guiar a busca em direção a soluções. Por fim apresentamos o problema de busca em árvore de Jogos.

3.1 Busca em profundidade

Na Figura 3.2 temos uma árvore, definida por um conjunto de nós. Podemos ver esta estrutura de dados como uma árvore de busca: cada caminho a partir da raiz representa um possível tentativa de solução do problema. Cada nó representa um estado do problema e cada arco representa um movimento. Uma busca parte do estado inicial, que é a raiz, e vai em direção a um estado objetivo. Uma folha representa um caminho sem saída, a partir do qual não temos mais movimentos para novos estados.

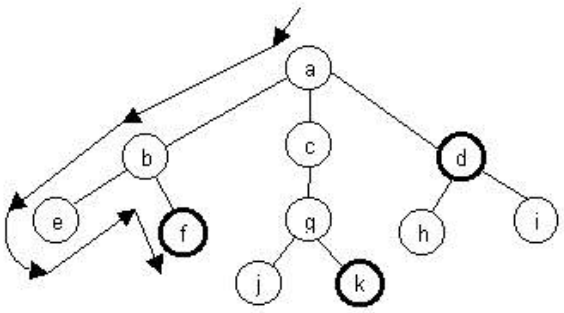


Figura 3.1: A estratégia de busca em profundidade: soluções encontradas na ordem: f, k, d.

Em muitos problemas as soluções estão sempre nas folhas. Mas podemos ter também soluções antes das folhas. Na Figura 3.2 temos três soluções: (d), (k) e (f); a solução (d) não é uma folha. Esta árvore é representada no código abaixo. As soluções são codificadas numa cláusula objetivo.

```
1 move(a,X):-X=b;X=c;X=d.
2 move(b,X):-X=e;X=f.
3 move(c,X):-X=g.
4 move(d,X):-X=h;X=i.
5 move(g,X):-X=j;X=k.
6 %
7 objetivo(S):-member(S,[f,k,d]).
```

Note que a árvore não é explicitamente construída como um termo. Pode nem ser necessário construir ou caminhar em toda a árvore para encontrarmos uma solução. Num problema de busca devemos evitar o caminhamento exaustivo. Queremos apenas alcançar um estado que é a solução do problema.

Seguem dois algoritmos para caminhamento em profundidade. O primeiro ignora o problema dos ciclos, já o segundo evita entrar num ciclo.

```
1 profundi(N,S):-profundi0(N,S).
2 profundi0(N,[N]) :- objetivo(N).
3 profundi0(N,[N|Sol1]) :- move(N,N1),profundi0(N1,Sol1).
4 %%
5 profundiH(N,H):-profundiH0(N,[N]/H).
6 profundiH0(N,H/H) :- objetivo(N).
7 profundiH0(N,Hi/Ho) :- move(N,N1), \+ member(N1,Hi),
8                         profundiH0(N1,[N1|Hi]/Ho).
```

Segue a execução duas versões. A primeira traz o caminho a partir da raiz, enquanto que a segunda traz o caminho na ordem inversa.

```
?- profundi(a,S).
S = [a,b,f] ;
```

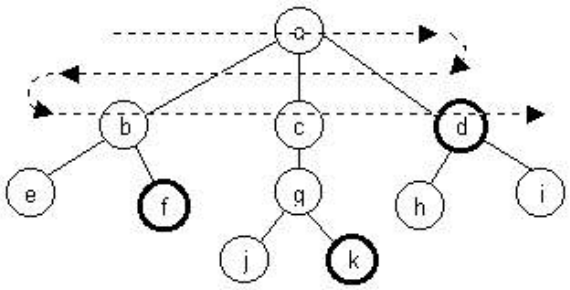


Figura 3.2: Estratégia de busca em largura: soluções encontradas na ordem: d, f, k.

```
S = [a,c,g,k] ;
S = [a,d] ;
no

?- profundiH(a,S).
S = [f,b,a] ;
S = [k,g,c,a] ;
S = [d,a] ;
no
```

Estes algoritmos trazem por retrocesso todas as soluções: trazer todas as soluções implica num caminhamento exaustivo. Em muitos casos só a primeira solução é necessária.

3.2 Busca em largura

Diferente da busca em profundidade, a busca em largura trata sistematicamente de todos os filhos de um nó, processando paralelamente todos os caminhos de cima para baixo. Portanto, quando a solução de um problema está próxima da raiz a busca em largura é mais eficiente que a busca em profundidade.

O predicado `largura`, abaixo, trata de modo sistemático todos os caminhos (vias) de cima para baixo. Eles são coletados em uma fila. Para isso, o predicado `estende` coleta todos os caminhos a partir de um nó; estes são incluídos no final da fila (implementada numa lista). A cada momento o primeiro nó da fila está sendo processado e cada novo caminho encontrado é inserido no final da fila.

Assim sendo, a estrutura de dados básica deste caminhamento é uma fila de vias (de caminhos a partir da raiz – representados na ordem inversa). Esta fila representa a fronteira da árvore já processada, que vem crescendo até incluir a primeira solução.

No processamento temos duas possibilidades:

- se o nó objetivo é a cabeça da primeira via da fila então termina a busca;
- senão, retiramos a primeira via da fila para ser estendida.

```
1 largura(X, Via):- largura0([[X]], Via).
2 largura0([[N|Via]|_], [N|Via]):- objetivo(N).
```

```
largura0([V|FilaI],  Via ):-
      estende(V, Fila), append(FilaI, Fila, Fila0),
      nl,write('Fila Total:'), write(Fila0),nl,
      largura0(Fila0, Via).
estende([N|Via], Fila):-
     bagof([NN, N|Via],( move(N,NN), not member(NN, [N|Via]) ), Fila),
     nl,write('Fila Estende:'), write(Fila),nl,!.
estende(Via, []).
```

O predicado `estende`, coleta os caminhos a partir de um determinado nó; no primeiro argumento do `bagof` especificamos os caminhos a serem coletados; no segundo argumento especificamos um predicado que determina os movimentos válidos.

```
?-bagof(X,move(a,X),B).
X = _ ,
B = [b,c,d]

?- bagof([X,a],move(a,X),B).
X = _ ,
B = [[b,a],[c,a],[d,a]]

?- bagof([X,b,a],move(b,X),B).
X = _ ,
B = [[e,b,a],[f,b,a]]
```

Segue a execução da busca em largura, onde usamos alguns comandos de escrita para monitorar o comportamento do algoritmo. Inicialmente, o `estende` acrescenta os três caminhos (vias) do nó (a) com os seus filhos (b, c, d). Num segundo passo, a via `[b,a]` é processada gerando as vias `[e,b,a]` e `[f,b,a]`. Note que sempre processamos o primeiro da fila e acrescentamos as novas vias no final da fila.

```
?-largura(a,S).
Fila Estende:[[b,a],[c,a],[d,a]]
Fila Total:[[b,a],[c,a],[d,a]]
Fila Estende:[[e,b,a],[f,b,a]]
Fila Total:[[c,a],[d,a],[e,b,a],[f,b,a]]
Fila Estende:[[g,c,a]]
Fila Total:[[d,a],[e,b,a],[f,b,a],[g,c,a]]
S = [d,a] ;

Fila Estende:[[h,d,a],[i,d,a]]
Fila Total:[[e,b,a],[f,b,a],[g,c,a],[h,d,a],[i,d,a]]
Fila Total:[[f,b,a],[g,c,a],[h,d,a],[i,d,a]]
S = [f,b,a] ;

Fila Total:[[g,c,a],[h,d,a],[i,d,a]]
Fila Estende:[[j,g,c,a],[k,g,c,a]]
```

```
Fila Total:[[h,d,a],[i,d,a],[j,g,c,a],[k,g,c,a]]
Fila Total:[[i,d,a],[j,g,c,a],[k,g,c,a]]
Fila Total:[[j,g,c,a],[k,g,c,a]]
Fila Total:[[k,g,c,a]]
S = [k,g,c,a] ;

Fila Total:[]
no
```

Profundidade vs largura

Em termos de resultados os dois tipos de busca retornam os mesmos três resultados, em ordens diferentes. Estas duas estratégias básicas de busca podem ser comparadas em termos de computabilidade e, recursos de memória e tempo.

Computabilidade. Em muitos espaços de busca, um caminho em profundidade pode ser infinito. Assim uma busca em profundidade pode não ser computável. No entanto, para um dado problema, se existe uma solução em um número finito de passos uma busca em largura necessariamente encontrará a solução.

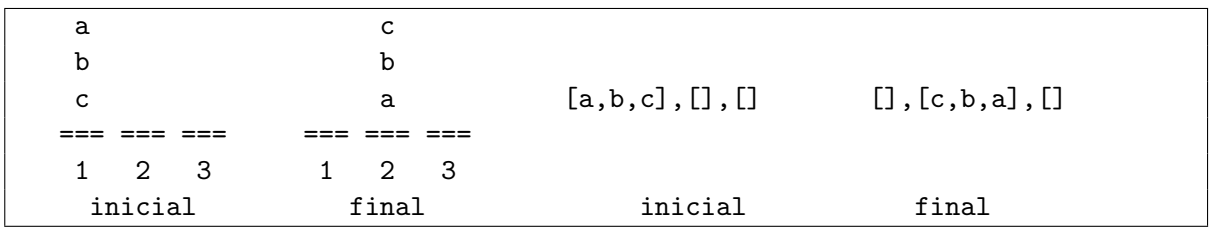
Espaço de memória e tempo. Numa busca em profundidade é suficiente armazenar uma pilha com todos os nós sendo visitados a partir da raiz. O tamanho máximo da pilha é a profundidade da árvore. Na busca em largura é necessário guardar todos as possíveis vias para serem processadas. Logo, em termos de espaço e também tempo de computação, usualmente, uma busca em profundidade é mais eficiente.

3.3 Espaço de busca

A árvore apresentada na discussão de busca em profundidade e largura, com uma dezenas de nós, serve apenas como exemplo didático de busca. Em problemas reais, as árvores de buscas podem ser objetos potencialmente infinitos. Estudaremos aqui dois pequenos problema práticos (problema dos blocos e das torres de Hanoi) para ilustrar a complexidade e tamanho de um espaço de busca.

O problema dos blocos

O problema dos blocos é enunciado assim: são três mesas (1, 2, 3) e três blocos empilhados (a, b, c) numa situação inicial. Queremos move-los da situação inicial para uma situação final, por exemplo, para outra mesa em uma outra ordem. A cada movimento é permitido mover apenas um bloco de um pilha para outra pilha (que pode ser vazia) em outra mesa. Queremos saber qual é a seqüência de movimentos que leva da situação inicial para a situação final?



Na modelagem de um problema devemos considerar três componentes:

- um estado inicial;
- um estado final;
- as regras para movimentação, de um estado para outro.

Estes três componentes são suficientes para se gerar um espaço de busca. Para o problema dos blocos representamos o estado inicial como uma lista com três sublistas, uma para cada mesa. Assim o estado inicial é `[[a,b,c],[],[]]` e o estao final é `[[],[c,b,a],[]]`. Nesta representação o predicado `move`, abaixo, movimenta um bloco de uma mesa para outra mesa; os blocos são movimentados do topo de uma pilha para o topo de outra; o topo corresponde a cabeça da lista. Em cada movimento uma das mesas permanece com o mesmo conteúdo.

```
1 move([X,Y,Z], [X1,Y1,Z1]):- Z=Z1,move0(X,Y,X1,Y1);
2                              Y=Y1,move0(X,Z,X1,Z1);
3                              X=X1,move0(Y,Z,Y1,Z1).
4 move0([X|Xs],Ys,  Xs,[X|Ys]):-!.
5 move0(Xs,[X|Ys],  [X|Xs],Ys).
6 %%
7 objetivo([[],[a,b,c],[]]).
```

Seguem alguns testes para o predicado `move`. O predicado `bagof` coleta todos os possíveis movimentos, a partir de uma dada posição.

```
?- move([[c,a,b],[],[]],Y).
   Y = [[a,b],[c],[]] ;
   Y = [[a,b],[],[c]] ;
   no
?- bagof(X,move([[c,b,a],[],[]],X),B).
   X = _ ,
   B = [[[b,a],[c],[]],  [[b,a],[],[c]]]

?- bagof(X,move([[],[],[c,b,a]],X),B).
   X = _ ,
   B = [[[c],[],[b,a]],  [[],[c],[b,a]]]
```

Segue a execução da busca em profundidade e em largura. Note que para este problema a busca em largura retornou uma solução bem menor que a busca em profundidade. Isto significa que a busca em profundidade considera vários movimentos que não são relevantes para se alcançar o estado da solução.

```
?- profundiH([[c,a,b],[],[]],S).
S = [[[],[a,b,c],[]],  [[a],[b,c],[]],  [[],[b,c],[a]],  [[b],[c],[a]],
[[b],[],[c,a]],[[],[b],[c,a]],[[],[c,b],[a]],[[c],[b],[a]],[[c],[a,b],[]],
[[],[a,b],[c]],[[a],[b],[c]],[[a],[],[b,c]],[[],[a],[b,c]],[[],[b,a],[c]],
[[b],[a],[c]],[[b],[c,a],[]],[[],[c,a],[b]],[[c],[a],[b]],[[c],[],[a,b]],
[[],[c],[a,b]],[[a],[c],[b]],[[],[a,c],[b]],[[],[b,a,c],[]],[[b],[a,c],[]],
[[a,b],[c],[]],  [[c,a,b],[],[]]]
```

```
?- largura([[c,a,b],[],[]],S).
S = [[[],[a,b,c],[]],  [[a],[b,c],[]], [[],[b,c],[a]], [[b],[c],[a]],
     [[a,b],[c],[]],  [[c,a,b],[],[]]]
```

Torres de Hanoi

O problema dos blocos é similar ao problema das torres de Hanoi (ver capítulo sobre animação de programas). A cada movimento movemos um disco de uma haste para outra, até se chegar numa solução.

Podemos usar o predicado `move` definido para o problema do blocos para mover os discos. Porém, no problema das torres de Hanoi devemos impor uma condição adicional ao movimento, para impedir um movimento de um disco maior sobre um disco menor. Para isso acrescentamos o predicado `valido` ao predicado `move` do problema dos blocos. Além disso é necessário modificar a cláusula objetivo, que define quando a busca atingiu o objetivo. Segue a codificação do estado de busca para o problema das torres de Hanoi.

```
1  move(X,Y):-move1(X,Y),valido(X).
2  move1([X,Y,Z], [X1,Y1,Z1]):- Z=Z1,move0(X,Y,X1,Y1);
3                                Y=Y1,move0(X,Z,X1,Z1);
4                                X=X1,move0(Y,Z,Y1,Z1).
5  move0([X|Xs],Ys,  Xs,[X|Ys]):-!.
6  move0(Xs,[X|Ys],  [X|Xs],Ys).
7  %%
8  objetivo([[],[1,2,3],[]]).
9  %%
10 valido([X,Y,Z]):-valido0(X),valido0(Y),valido0(Z).
11 valido0(X):-member(X,[ [],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3] ]).
```

Abaixo temos duas soluções para o problema das torres.

```
?- profundiH([[1,2,3],[],[]],S).
S = [[[],[1,2,3],[]],[[1],[2,3],[]],[[],[2,3],[1]],[[2],[3],[1]],[[2],[1,3],[]],[[],
[1,3],[2]],[[1],[3],[2]],[[],[3],[1,2]],[[3],[],[1,2]],[[3],[1],[2]],[[2,3],[1],
[]],[[1,2,3],[],[]]] ;

S = [[[],[1,2,3],[]], [[1],[2,3],[]], [[],[2,3],[1]], [[2],[3],[1]], [[2],[1,3],[]],
[[],[1,3],[2]], [[1],[3],[2]], [[],[3],[1,2]], [[3],[],[1,2]],
[[3],[1],[2]], [[2,3],[1],[]], [[2,3],[],[1]], [[1,2,3],[],[]]]
```

Note que a solução para o problema das Torres de Hanoi apresentada no capítulo sobre animação de programas não faz nenhum movimento desnecessário para atingir o objetivo. Isto é o algoritmo retorna uma solução ótima.

Na modelagem deste capítulo, como um problema de busca, podem ser feitos inúmeros movimentos desnecessários. Gera-se uma solução, mas, tipicamente, não uma solução ótima. Para direcionar a busca, para trazer uma solução ótima devemos incluir regras de heurística. Este tema é estudado na próxima seção.

Exercício 3.1 Generalize a solução apresentada para N discos.

Exercício 3.2 Anime a solução, a partir da lista de movimentos gerada.

Exercício 3.3 Resolva o problema *do lobo, do cordeiro e do pasto.* Um fazendeiro está voltando dos trabalhos trazendo um lobo, um cordeiro e um feixe de pasto. Para chegar em casa ele tem que atravessar um rio, numa canoa. O problema é que na canoa só tem lugar para dois, o fazendeiro e um dos três. Se ele leva o lobo, o cordeiro vai comer o pasto; se ele leva o pasto o lobo come o cordeiro. O fazendeiro precisa saber em que seqüência deve transportar um objeto (lobo, cordeiro, pasto) por vez, para que ninguém coma ninguém.

Crie um modelo, similar ao das torres de Hanoi, para este problema. São três objetos e duas posições (lado esquerdo e direito do rio). Na situação inicial todos estão num lado, na final todos estão no outro lado. △

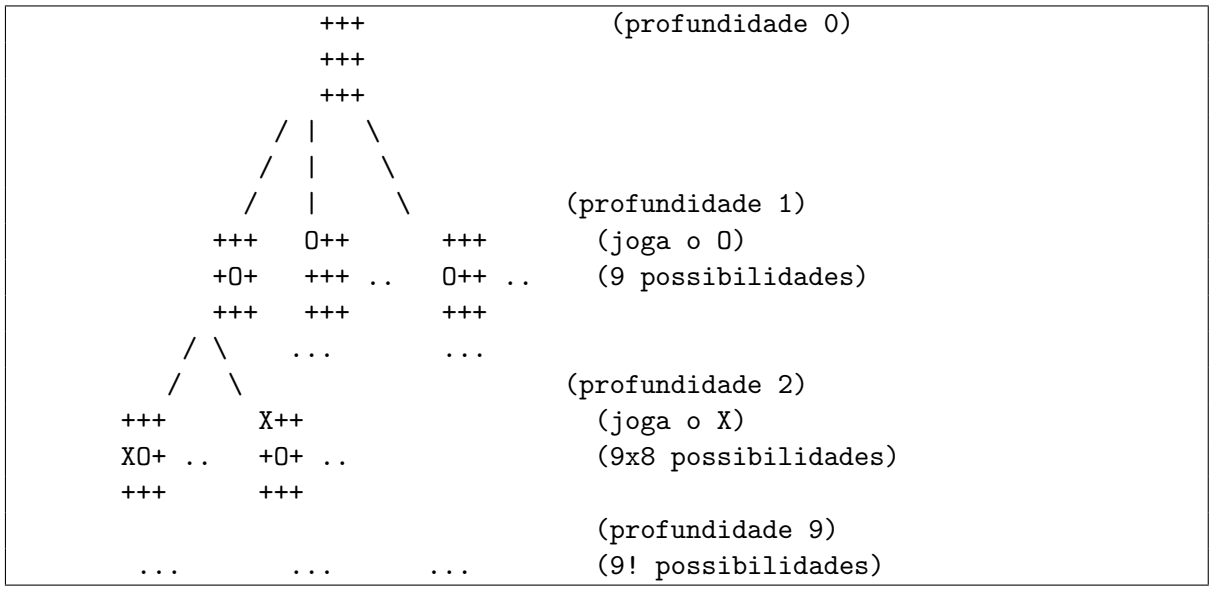
3.4 Busca em árvore de jogos

Uma das aplicações para os métodos de busca é a solução de jogos. Um jogo pode ser resolvido (a escolha da próxima jogada) por uma árvore de busca especializada chamada de árvore de jogo.

Aqui apresentaremos o problema de busca em árvores de jogos para dois jogadores.

Árvore de busca para jogos

Um árvore de jogo é uma estrutura onde cada nó representa um estado do jogo. Por exemplo, abaixo, representamos alguns nós de uma árvore para o jogo da velha (que foi apresentado no capítulo sobre animação de programas).



O problema da computação de árvores

Numa árvore para o jogo da velha a raiz é o tabuleiro vazio. Assim, temos 9 possíveis jogadas para o jogador que inicia o jogo: uma no meio; quatro nos cantos e quatro nas posições intermediárias. Para o segundo nível temos 72 (9x8) posições. No terceiro nível 504 (9x8x7).

Se toda a árvore for construída temos (9!+8!+..2!+1!) posições que são (362880+40320+5040+ ...) que são mais de 400 mil nós.

Devemos descontar deste números as possibilidades que levam um final de jogo antes das nove jogadas. Por exemplo, se ambos os jogadores jogam em duas linhas diferentes sem se defender, após cinco jogadas o jogador que iniciou completa três marcações na mesma linha ganhando o jogo, e o adversário teve apenas duas marcações; assim o jogo termina em 5 jogadas, ver abaixo. Por outro lado, numa árvore de jogo, diferentes caminhos podem levar a uma mesma situação do jogo: no jogo abaixo que termina em cinco jogadas, o jogador bola pode ter iniciado no meio, o na lateral direita – para várias situações iniciais temos uma mesma situação final. Um processo de construção de árvores de jogos eficiente deveria compartilhar (ou remover) situações comuns, reduzindo o tamanho da árvore do jogo.

```
OOO
XX+
+++
```

O total do número de nós para uma árvore de jogo, como o Jogo da Velha, ilustra o tamanho de uma árvore de jogo. Até mesmo para um jogo com um tabuleiro 3x3 e com poucas possibilidades de próxima jogada temos um valor alto: 400 mil nós. Apesar de significativo, 400 mil é um número facilmente computável em qualquer computador pessoal.

Dois fatores contribuem no tamanho de uma árvore de jogo:

- o número de jogadas necessária para completar um jogo determina a profundidade da árvore;
- o número de possibilidade da próxima jogada, determina com que fator a árvore cresce em largura a cada nível.

Normalmente quanto maior o tabuleiro do jogo, maior é o número de possibilidades para se executar a próxima jogada. Por exemplo, para o xadrez, num tabuleiro de 8x8, temos em média 35 possíveis próximas jogadas e podemos ter em média 50 jogadas para cada jogador até se concluir o jogo. Portanto, o tamanho da árvore é de $100^3 5$.

Movimentos em árvores de jogos

Numa árvore de jogo, um jogo corresponde a um caminho da raiz até uma folha; um movimento é uma jogada; os movimentos são intercalados, um para cada jogador. No exemplo do jogo da velha, apresentado acima, numa profundidade par a bola joga e numa impar a cruz joga.

O jogo pode terminar antes de alcançar a folha da árvore com todos os possíveis movimentos. Abaixo, mostramos um jogo que termina duas jogas antes do tabuleiro estar totalmente preenchido. Neste jogo, o jogador bola iniciou jogando no centro do tabuleiro. O jogador cruz jogou numa posição intermediara. Nesta situação do jogo, a partir da terceira jogada, existem duas principais possibilidades de movimento para o jogador bola:

- Uma jogada num dos cantos levará a uma vitória, como exemplificado no galho superior.
- Uma jogada numa posição intermediaria não garante a vitória. Portanto, existem duas possibilidades de jogadas intermediarias:

– uma jogada que não ameaça o adversário (a posição intermediaria á direita);
– a jogada que ameaça o adversário (a posição intermediaria superior ou inferior).

Abaixo, ilustramos uma vitória para bola em sete movimentos: nesta situação a jogada no canto, cria uma dupla ameaça que leva a vitória.

```
profundidade
(2)     (3)    (4)    (5)    (6)    (7)
        O++    O++    O+O    O+O    OOO
        XO+----XO+----XO+----XO+----XO+  vitória do (O)
     /+++    ++X    ++X    X+X    X+X
     /

+++/  +++
XO+ ----XOO  ... não ameaça o (X)
+++ \   +++
     \
      \ +O+
        XO+  ... ameaça o (X)
        +++
```

Neste exemplo a escolha da melhor jogada contabiliza (ou pensa) 5 jogadas na frente (até a folha). A inteligência do algoritmo depende da possibilidade de analisar vários níveis de profundidade numa árvore. Quanto maior a profundidade mais inteligente é o algoritmo.

Na mesma situação exemplificada, a escolha de uma posição intermediaria leva ao empate, salvo erros triviais do adversário (por exemplo, não se defender quando é atacado).

3.4.1 Minimax

O *minimax* é um algoritmo fundamental no estudo de árvores de jogos. Este algoritmo constrói uma árvore completa para o jogo. Como apresentamos acima, até mesmo para o jogo da velha, uma árvore de jogos completa tem mais de 400 mil nós. Aqui, apresentaremos o minimax com um exemplo de uma pequena árvore hipotética com uma dezena de nós.

Para um jogo simples (terminado com vitória ou empate) podemos usar a seguinte definição de árvore de jogo:

- Cada nível da árvore é associado a um jogador (chamados MAX e MIN);
- Um nó folha MAX (associado ao jogador MAX) é avaliado com a função de utilidade: +1 vence, -1 perde, 0 empata;
- Um nó folha MIN é avaliado com a função de utilidade: -1 vence, +1 perde, 0 empata;
- Cada nó intermediário MIN é avaliado como o menor valor de seus filhos;
- Cada nó intermediário MAX é avaliado como o maior valor de seus filhos.

Tecnicamente, a estratégia de avaliação dos nós intermediários assume que ambos os jogadores sempre fazem a melhor jogada, o que na prática, num jogo com pessoas raramente acontece.

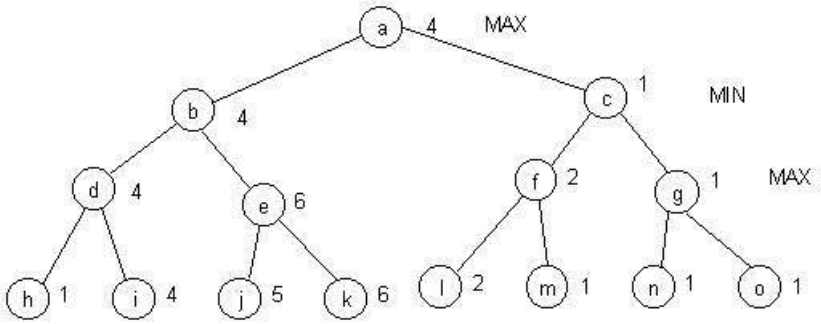


Figura 3.3: Uma árvore minimax para um jogo; baseada na função de utilidade.

A Figura 3.3 apresenta um exemplo de uma árvore de jogo. Nesta árvore, com as regras acima, o algoritmo minimax determina a melhor jogada em cada nó. Inicialmente só os nós folha possuem o valor dado pela função de utilidade. O processo de avaliação dos valores intermediários acontece de debaixo para cima, até chegar a raiz da árvore. Por exemplo, o valor 4 para o nó (d,MAX) é resultado da escolha do maior entre $\{(h,1),(i,4)\}$; o valor 4 no nó (b,MIN) é escolhido como o menor dos valores de seus filhos, $\{(d,4),(e,5)\}$; e assim por diante.

A árvore Figura 3.3 é representada em Prolog como um conjunto de fatos, um para cada arco. Cada nó folha está associado ao valor da função de utilidade. Os nós intermediários possuem um valor que indica qual dos filhos foi escolhido como a melhor jogada.

```
1  move(a,b).
2    move(b,d).
3        move(d,h). move(d,i). val(h,1). val(i,4).
4    move(b,e).
5        move(e,j). move(e,k). val(j,5). val(k,6).
6  move(a,c).
7    move(c,f).
8        move(f,l). move(f,m). val(l,2). val(m,1).
9    move(c,g).
10       move(g,n). move(g,o). val(n,1). val(o,1).
11 %%
12 max(X):-!,member(X,[a,d,e,f,g]).
13 min(X):- \+ max(X).
14 terminal(Pi) :- \+ move(Pi,_).
```

Basicamente, o `minimax` executa um percurso recursivo em profundidade sobre a árvore. Na volta, em cada nó intermediário é escolhida a melhor jogada, o valor mínimo ou máximo, conforme o nível correspondente ao jogador (MIN, MAX).

O predicado `melhor` escolhe o melhor par, (nó/valor), da lista dos filhos de um nó; ele usa o predicado `melhorPar` que seleciona o melhor entre dois pares, respeitando o nível de profundidade da árvore associado a MIN ou MAX.

```
1  minimax(Pi, Po/Val ):- terminal(Pi),!,val(Pi,Val), Po=Pi.
2  minimax(Pi, Po/Val ):-
```

```
3     bagof(X, move(Pi,X), List),
4     melhor(List, Po/Val),!.
5  %%
6  melhor([Po],  Po/Vo):- minimax(Po, _/Vo),!.
7  melhor([P|Ps],   Po/Vo):-
8         minimax(P, _/V),
9         melhor(Ps, P2/V2),
10        melhorPar( P/V, P2/V2, Po/Vo ).
11 %%
12 melhorPar(P0/V0, P1/V1, P0/V0):-min(P0), V0>V1,!;max(P0), V0<V1,!.
13 melhorPar(_/_,   P1/V1, P1/V1).
```

Seguem alguns testes para os valores da árvore exemplificada acima. Estes valores podem ser conferidos na árvore da Figura 3.3, onde os nós são anotados com os valores já calculados. Em cada nó o minimax escolhe a a melhor jogada, o valor mínimo se o nó for MIN e o valor máximo caso contrário.

```
?- minimax(a,P).
   P = par(b,4)
?- minimax(b,P).
   P = par(d,4)
?- minimax(c,P).
   P = par(g,1)
?- minimax(g,P).
   P = par(o,1)
```

3.4.2 Minimax com poda Alfa-Beta

O problema com o minimax é que ele necessita avaliar a árvore completa de um jogo para tomar a decisão sobre a melhor jogada. Na prática uma árvore de jogo para jogo complexo como o xadrez é uma estrutura enorme: são 100^{35} nós. Portanto, não é viável representar e computar uma árvore com estas dimensões.

Em face a este problema, foram desenvolvidos dois aprimoramentos sobre o algoritmo minimax:

- a poda alfabeta — otimiza o cálculo da árvore de jogo; somente uma parte do nós são avaliados;
- o corte na profundidade — o uso de funções de avaliação para cortar a avaliação de nós a partir de determinada profundidade; limita o exame do número de jogadas à frente.

O algoritmo Alfa-Beta

Seque o algoritmo alfabeta. Os detalhes técnicos são apresentados a seguir.

```
1  alfabeta(M/M, Pi, A/B,  Po/Vo):- !, val(Pi,Vo), Po=Pi.
2  alfabeta(N/M, Pi, A/B,  Po/Vo):- terminal(Pi),!,val(Pi,Vo), Po= Pi.
3  alfabeta(N/M, Pi, A/B,  Po/Vo):-
```

```
4    bagof(X, move(Pi,X), List),
5    melhorAB(N/M, List, A/B,  Po/Vo).
6  %%
7  melhorAB(N/M, [P|Ps],  A/B,  Po/Vo):-
8    N1 is N+1,
9    alfabeta(N1/M, P,  A/B,     _/V),
10   cortaAB(N/M, Ps,  A/B,  P/V, Po/Vo).
11 %%
12 cortaAB(N/M, [],  _/_,  P/V,   P/V):-!.
13 cortaAB(N/M, _,   A/B,  P/V,   P/V):- min(P), V>B,!; max(P), V<A,!.
14 cortaAB(N/M, Ps,  A/B,  P/V,  Poo/Voo):-
15   novoAB(A/B,  P/V,   Ao/Bo),
16   melhorAB(N/M, Ps, Ao/Bo,  P1/V1),
17   melhorPar(P/V,    P1/V1,  Poo/Voo).
18 %%
19 novoAB(A/B,  P/V,  V/B ) :- min(P), V>A,!.
20 novoAB(A/B,  P/V,  A/V ) :- max(P), V<B,!.
21 novoAB(A/B,  _/_,  A/B ).
```

O processo de busca em profundidade do alfabeta termina em três casos:

- Quando é identificado um ramo que não influencia o resultado, este é cortado (otimização);
- Quando ele atinge a profundidade estipulada. Neste caso a *função de avaliação* é aplicada ao estado do jogo representado no nó;
- Quando é atingida a fronteira da árvore; isto é quando alguém ganha o jogo, oo quando o jogo empata; a *função de utilidade* é aplicada.

O predicado `alfabeta(N/M, Pi, A/B,  Po/Vo)` possui quatro parâmetros; os três primeiros são de entrada e o último é de saída. O par `N/M` controla a profundidade, quando `N=M` o processo pára (segundo caso). O `Pi` é a posição de entrada. O par `A/B` denota o alfa/beta; inicialmente assume-se um alfa igual ao um valor mínimo e beta igual a um valor máximo; quando os primeiros nós MAX e MIN são processados estes valores são substituídos, por valores reais que fazem o par alfa/beta convergir para o zero. Os valores alfa/beta descem pela árvore para controlar o corte. O par `Po/Vo`, posição/valor (escolhido), sobe pela árvore determinando o valor de cada nó. Abaixo detalhamos este processo.

A poda alfabeta

A árvore Figura 3.4 ilustra a poda alfa-beta. Por exemplo, na avaliação em profundidade, de baixo para cima, o nó (e) não precisa ser totalmente avaliado, pois, o nó (d) já foi avaliado com valor 4 e a avaliação parcial do nó (e) já escolhe o valor 5; o nó (b,MIN) escolhe o menor, e o valor 5 já é maior que o 4. De modo similar, quando já avaliamos parcialmente o nó (c,MIN) temos o valor 2 que não influencia na computação do (a,MAX); que escolherá o máximo, o valor 4 do (b).

Mais formalmente, temos que:

- $MAX(\alpha,\ MIN[\ V|...]) = \alpha\ se\ V < \alpha$

- $MIN(\beta,\ MAX[\ V|...]) = \beta\ se\ V > \beta$

Neste caso corta-se a avaliação dos irmãos do nodo (`cortaAB/5`). Estas regras instanciadas no exemplo da Figura 3.3 resultam em:

- `MAX[a/alfa=4, MIN[c/2 | ...]] = 4 se 2<4`
- `MIN[d/beta=4, MAX[j/5 | ...]] = 4 se 5>4`

O detalhe está na inicialização e atualização destes valores limites. O α é o limite mínimo em nós MIN; e o β é o limite máximo em nós MAX. Num determinado nó da árvore a busca se limita a processar nós irmãos que estejam dentro destes limites (alfa <= X <= beta). No algoritmo, estes limites são inicializados com dois valores estremos, um negativo para α alfa e um positivo para β. Durante o processamento de um nó, num mesmo nível da árvore, a cada valor calculado por `alfabeta`, estes limites α *e* β são atualizados com as regras(`novoAB/3`):

- num nó pai MIN, se um filho MAX possui um valor menor que beta então beta recebe o seu valor; pois, MIN escolhe o menor;
- num nó pai MAX, se um filho MIN possui um valor maior que alfa então alfa recebe o seu valor; pois, MAX escolhe o maior;

Se a profundidade de uma árvore é d, e temos n movimentos possíveis em cada nó, o algoritmo minimax tem que examinar $O(n^b)$ nodos. Com a poda alfa-beta, no melhor do casos (sempre escolhendo-se a melhor jogada) o corte eqüivale a remoção de metade da profundidade, o que resulta em $O(n^{b/2})$ – em outras palavras, se $n^b = N$ então $n^{b/2} = \sqrt{N}$. O possível número de nós que não são examinados e significativo. Por exemplo, para o jogo da velha $\sqrt{400000} = 632$ – isto seria no melhor dos casos; no pior dos casos a redução é pequena. Portanto, para problemas reais esta otimização ainda é insuficiente.

Corte na profundidade

Uma técnica complementar é o corte em profundidade, onde corta-se a avaliação a partir de uma determinada profundidade da árvore. Por exemplo, num jogo de xadrez, um campeão pensa até 40 jogadas na frente (profundidade). Se não temos recursos computacionais para "pensar" 40, podemos programar o algoritmo para pensar 10 jogadas na frente. Neste caso os valores da parte da árvore que não foram computados devem ser estimados, com o uso de uma função de avaliação.

Se temos uma função avaliação para qualquer nó da árvore podemos chamar o algoritmo para avaliar qualquer nível de profundidade. Em princípio, quanto menor a profundidade, menos inteligente é a solução gerada. Este método é também usado para definir os níveis de dificuldade nos jogos (iniciante, médio, profissional).

Uma *função de avaliação* define um "chute", para uma subárvore não avaliada, usando apenas a informação do nó raiz até o nó sendo avaliado. A qualidade de uma escolha depende da qualidade desta "chute". Para o exemplo da árvore da Figura 3.3 definimos os valores abaixo.

```
1 val(a,3). val(b,1). val(c,5).
2 val(d,6). val(e,3). val(f,4). val(g,2).
```

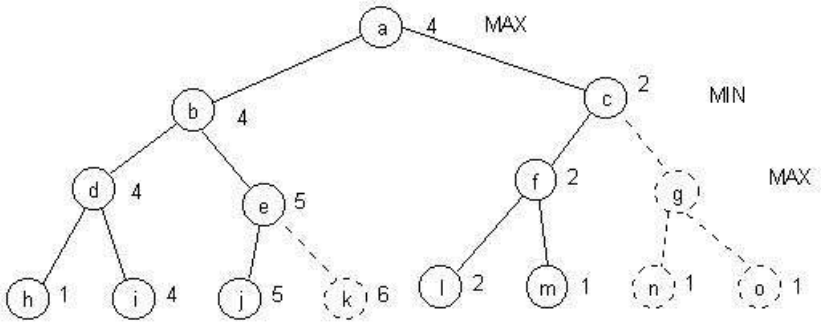


Figura 3.4: Uma árvore minimax com poda alfa-beta; os nós pontilhados não são avaliados.

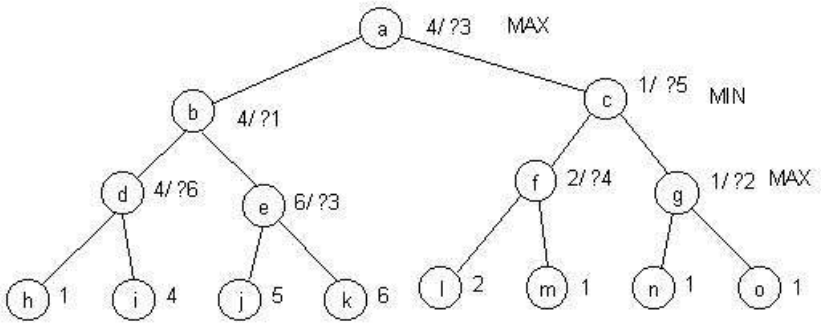


Figura 3.5: Uma árvore minimax com poda alfa-beta; com nós valorados também por uma função de avaliação; o símbolo de interrogação indica que o valor é um chute (não é preciso como o valor do minimax).

A Figura 3.5 apresenta uma árvore onde os nós internos estão com os valores calculados pelo minimax e também anotados com os valores da função de avaliação `val/2`, definida abaixo. Note que este mesmo predicado `val/2` foi usado como função de utilidade, para definir o valor para os nós folha.

Segue uma lista de testes para diferentes combinações de posições e/ou profundidades para a árvore da figura acima. Note que, dependendo da profundidade, a avaliação de um nó pode trazer valores diferentes, por exemplo, avaliar (a) com profundidade três resulta no (b), porém avaliar (a) com profundidade um resulta em (c).

```
?- alfabeta(0/3, a,-100/100, P/V).
   P = b , V = 4
?- alfabeta(0/3, b,-100/100, P/V).
   P = d , V = 4
?- alfabeta(0/1, a,-100/100, P/V).
   P = c , V = 5
?- alfabeta(0/3, c,-100/100, P/V).
   P = g , V = 1
```

Numa aplicação prática deste algoritmo enfrenta-se vários problemas técnicos, tais como o tamanho de uma árvore para um jogo e a dificuldade de se estabelecer funções de avaliação

para os nós. Diante disso, é difícil desenvolver programas competitivos, para jogos não triviais como xadrez, "backgammon" e "go". Para o "go", que é jogado num tabuleiro de 19x19 (bem maior que o do xadrez) existe um prêmio de US$ 1.6 milhões de dólares (em 2001) para o primeiro programa de computador que combater um dos campeões mundiais (ver detalhes em http://www.usgo.org/computer/icgc.html)[1].

Para concluir o tema sobre árvore de busca para jogos, ilustramos o uso do algoritmo alfabeta para o jogo da velha.

3.4.3 Um exemplo de uso: Jogo da Velha

O jogo da velha já foi apresentado no capítulo sobre animação de programas. Aqui assumimos a mesma representação para o tabuleiro, com a enumeração dada abaixo.

```
  1|2|3   x|x|
  -----   -----
  4|5|6   o|x|o
  -----   -----
  7|8|9    | |o
```

Na versão com a busca alfabeta permitimos ambos iniciar o jogo: o computador ou o adversário (oponente). Abaixo, temos as inicializações para o jogo, onde a possibilidade do oponente começar é deixada como comentário.

```
1  %%
2  %% Computador Inicia:
3  %%         tab(Lista,            Profund).
4  inicializa(tab([*,*,*, *,*,*, *,*,*], 0),computador).
5  computador(o).
6  profundidade(2).
7  %%
8  %% Oponente Inicia:
9  %%
10 % computador(x).
11 % inicializa(tab([*,*,*, *,*,*, *,*,*], 0),oponente).
12 % profundidade(3).
```

Assumimos que o nó raiz da árvore, com profundidade 0, é associado ao jogador MAX; assim, todos os valores pares na profundidade da árvore são do jogador MAX e os impares do jogador MIN; isto é, num valor par o MAX joga e num valor impar o MIN joga.

O tabuleiro é representado por `tab(Pos9, Prof)`, onde `Pos9` é a lista de nove posições e `Prof` é a profundidade da jogada. Assumimos que sempre o MAX joga com bola e o MIN joga com cruz. O predicado `x2o(P,C)`, dado um tabuleiro ou o valor da profundidade, retorna em `C` o valor cruz ou bola, respectivamente para MIN ou MAX.

[1]Caso o URL tenha mudado, para encontrar o novo URL, tente pesquisar as palavras chave: *Computer Go Tournaments/ICGC/prize/million/professional go/usgo*

```
1  x2o(P,C):-max(P),!,C=o.
2  x2o(P,C):-min(P),!,C=x.
3  %%
4  proxJogar(o,x):-!.
5  proxJogar(x,o):-!.
6  %%
7  max(tab(L,M)) :-!,max(M).
8  max(M) :- integer(M),0 is M mod 2.
9  min(X) :- \+  max(X).
10 profund(tab(T,P),P).
11 %%
12 vaziaN(tab(L,M), N):- n_member(N,L,*).
13 %%
14 n_member(1,[X|Xs],X).
15 n_member(N,[X|Xs],Y) :- n_member(N1,Xs,Y), N is N1+1.
16 %%
17 move(TABi,TABo):- vaziaN(TABi,N),move(N,TABi,TABo).
18 move(N,tab(L,P),tab(L1,Q)):-x2o(P,C),trocaN(N,C,L,L1),Q is P+1.
19 %%
20 troca(X, [],[]):-!.
21 troca(X, [*|Y], [X|Y1]):-!, troca(X,Y,Y1).
22 troca(X, [H|Y], [H|Y1]):-!, troca(X,Y,Y1).
23 %%
24 trocaN(1,C,[_|L], [C|L]):-!.
25 trocaN(N,C,[X|Li],[X|Lo]):-N1 is N-1,trocaN(N1,C,Li,Lo).
```

A árvore do jogo é construída a partir da posição inicial com o predicado `move/2` que gera todos os movimentos válidos. São usados também os predicados:

- `vaziaN` - retorna uma posição livre;
- `troca` - substitui todas as posições livres por uma marcação;
- `trocaN` - substitui a posição N por uma marcação;

Função de utilidade e avaliação

A parte essencial para o bom funcionamento do algoritmo alfabeta é a descoberta de uma função de utilidade e de outra função de avaliação. Como função de utilidade, numa folha onde alguém vence, assumimos:

- se a folha MIN vence atribuímos valor 1000 como valor de utilidade;
- se a folha MAX vence atribuímos valor -1000 como valor de utilidade.

Note que queremos passar estes valores para cima na árvore. Se a folha é MIN o pai dela é MAX e escolherá o maior valor, por isso temos que atribuir um valor positivo. E vice-versa.

Devemos também nos preocupar com a função de avaliação. Para o jogo da velha uma boa função de avaliação deve indicar quando estamos sendo ameaçados, e neste caso não tem sentido buscar uma jogada de ataque, mas sim parar a busca e nos defender. Portando codificamos:

- se na folha MIN estamos sendo ameaçados retornamos 100;
- se na folha MAX estamos sendo ameaçados retornamos -100;

Note que não existe uma grande diferença entre a função de utilidade e a função de avaliação. Em princípio a função de utilidade é usada só nos nós folha da árvore, quando um jogador vence ou quando o jogo empata. Porém, quando estamos ameaçados, se nós não nos defendermos, nós perderemos. Portanto, numa ameaça estamos quase na folha, mas com uma jogada de defesa o jogo pode continuar por algum tempo. Neste contexto, as funções de avaliação e utilidade se confundem. Na prática devemos impor uma escala de valores: por exemplo, 1000 para vencer; 100 para se defender; etc. Neste caso priorizamos o vencer.

A função de avaliação também deve dar uma escala de valores para três tipos de jogadas em *posições seguras* (não ameaçados):

- (maior) no meio do tabuleiro (posição 5);
- (médio) num dos cantos (posições 1, 3, 7, 9);
- (menor) nas restantes (posições 2, 4, 6, 8).

Esta estratégia é codificada com o predicado `nVence(X,TAB,V)` que conta quantas possibilidades existem para um jogador fazer três marcas alinhadas, a partir de um dado tabuleiro `TAB`. Antes de contar as posições vazias são preenchidas como bola o cruz conforme o valor de jogador. Num nó MAX retornamos um valor negativo e num nó MIN um valor positivo.

```
1  val(P,  1000):- min(P), x2o(P,J),vence(P,J),!.
2
3  val(P, -1000):- max(P), x2o(P,J),vence(P,J),!.
4  val(P, -100 ):- max(P), x2o(P,C),proxJogar(C,CN),vence(P,CN),!.
5  val(P,  100 ):- min(P), x2o(P,C),proxJogar(C,CN),vence(P,CN),!.
6  val(P, V):- max(P), x2o(P,C),proxJogar(C,CN),
7              nVence(C,P,V1),nVence(CN,P,VN), V is V1 - VN,!.
8  val(P, V):- min(P), x2o(P,C),proxJogar(C,CN),
9              nVence(C,P,V1),nVence(CN,P,VN),V is VN - V1,!.
10 %%
11 nVence(X,TAB,V):-TAB=tab(L,_),troca(X,L,L1),nVence0(X,L1,V).
12 nVence0(X,L,V):- bagof(L,vence(tab(L,_),X),B),!,length(B,V).
13 nVence0(_,L,0).
14 %
15 vence(tab([X,X,X, _,_,_, _,_,_],_),X).
16 vence(tab([_,_,_, X,X,X, _,_,_],_),X).
17 vence(tab([_,_,_, _,_,_, X,X,X],_),X).
18 vence(tab([X,_,_, X,_,_, X,_,_],_),X).
19 vence(tab([_,X,_, _,X,_, _,X,_],_),X).
20 vence(tab([_,_,X, _,_,X, _,_,X],_),X).
```

```
21 vence(tab([X,_,_, _,X,_, _,_,X],_),X).
22 vence(tab([_,_,X, _,X,_, X,_,_],_),X).
23 %%
24 game_over(tab(L,Y),o):-vence(tab(L,_),o).
25 game_over(tab(L,Y),x):-vence(tab(L,_),x).
26 %%
27 game_over(TAB,empate):- nVence(o,TAB,V),V=0,
28                          nVence(x,TAB,V),V=0.
29 terminal(Tab) :- game_over(Tab,_),!.
30 terminal(Tab) :- \+ move(Tab,_), !.
```

Segue um exemplo de jogo. Sabemos que quando um jogador inicia na posição do meio, e o adversário não joga nos cantos, o adversário perde. Abaixo ilustramos a inteligência do computador vencendo o jogo.

```
?-joga.
     +++
     +o+      tab([*,*,*,*,o,*,*,*,*],1)
     +++
Faca um movimento:2.
     +x+
     +o+      tab([*,x,*,*,o,*,*,*,*],2)
     +++
Computador:
     +x+
     +o+      tab([*,x,*,*,o,*,*,*,o],3)
     ++o
Faca um movimento: 1.
     xx+
     +o+      tab([x,x,*,*,o,*,*,*,o],4)
     ++o
Computador:
     xxo
     +o+      tab([x,x,o,*,o,*,*,*,o],5)
     ++o
Faca um movimento: 7.
     xxo
     +o+      tab([x,x,o,*,o,*,x,*,o],6)
     x+o
Computador:
     xxo
     +oo      tab([x,x,o,*,o,o,x,*,o],7)
     x+o
Sinto! Mas eu, o computador, venci.
```

O algoritmo principal do jogo é o mesmo que foi apresentado no capítulo sobre animação de programas. A única diferença é que aqui quem escolhe a próxima jogada é o algoritmo alfabeta.

As regras heurísticas que direcionam a busca para a descoberta da próxima jogada se resumem na função de avaliação e utilidade.

```
joga :-  inicializa(Pos, Jogador),
         mostra_jogo(Pos, Jogador),
         joga(Pos, Jogador).
%%
joga(Pos, Jogador):- game_over(Pos,Result),!,wrtln(Result).
joga(Pos, Jogador):- selecionaMov(Pos,Jogador,NPos),!,
                     mostra_jogo(NPos, Jogador),!,
                     prox_jogar(Jogador, Oponente), !,
                     joga(NPos, Oponente).
prox_jogar(computador,oponente).
prox_jogar(oponente,computador).
%%
selecionaMov(Pos,computador, NPos) :-
     nl, write('Computador:'),nl,
     profundidade(M),
     alfabeta(0/M, Pos,-100/100, NPos/V).
%%
selecionaMov(Pos ,oponente,NPos) :-
   nl, write('Faca um movimento:'),nl,
   read(N),((  vaziaN(Pos,N),move(N,Pos,NPos),!
             ; write('Erro: Posicao invalida'),nl,!,
               selecionaMov(Pos ,oponente,NPos)  )).
%%
wrtln(empate):-!, write(['O jogo empatou']),nl.
wrtln(X):- computador(X), !, write('Sinto! Mas eu, o computador, venci.']),nl.
wrtln(X):- \+ computador(X), !, write('Voce venceu! Parabens !']),nl.
%%
mostra_jogo(TAB,J):- nl, write(TAB),nl,mostra_jogo0(TAB,J).
mostra_jogo0(tab([X1,X2,X3,X4,X5,X6,X7,X8,X9],_),Jogador) :-
  tab(4),write0(X1,X2,X3),nl,
  tab(4),write(''),nl,
  tab(4),write0(X4,X5,X6),nl,
  tab(4),write(''),nl,
  tab(4),write0(X7,X8,X9),nl.
write0(X1,X2,X3):- writeX(X1),write('|'),writeX(X2),write('|'), writeX(X3).
writeX(X):- X='*'->write(' ');write(X).
```

Exercício 3.4 Use um comentário nas instruções que fazem com que o computador inicia o jogo. Remova o comentário das instruções que inicializam o jogo para o oponente. Rode o programa e examine a inteligência do programa. Quais os melhoramentos que podem ser feitos no jogo em geral?

Referências Bibliográficas

[1] H. Ait-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Cambridge, 1991, (also in the Web).

[2] I. Brakto, *Prolog Programming for Artificial Intelligence*, Second Edition, Addison-Wesley Publishing Company. 1990.

[3] M. A. Casanova, F. A. C. Giorno e A. L. Furtado, *Programação em Lógica e a Linguagem Prolog* Edgar Blücher Ltda, Rio de Janeiro, 1987.

[4] W. F. Clocksin e C. S. Mellish, *Programming in Prolog* Springer-Verlag, 4th edition, 1994.

[5] M. A. Covington, D. Nute e A. Velino, *Prolog Programming in Depth*, Prentice Hall, New Jersey, 1997.

[6] M. A. Covington, *Natural Language Processing for Prolog Programmers*, Prentice Hall, New Jersey, 1994.

[7] P. Deransart, A. Ed-Dbali e L. Cervoni, *Prolog: The Standard – Reference Manual* Springer, Berlin, 1996.

[8] C. J. Hooger, *Essentials of Logic Programming*, Oxford University Press, Oxford, 1990.

[9] S. J. Russell e P. Norvig, *Artificial Intelligence: A modern approach*, Prentice Hall, New Jersey, 1995.

[10] L. Sterling e E. Shapiro, *The Art of Prolog*, The MIT Press, Cambridge, 1986.